Artificial Neural Network Simulator

ECE586 Final Project

Portland State University
Spring 2016
Saroj Bardewa & Conor O'Connell

I. Introduction and Motivation

This document is the report for the final project of ECE 586, Computer Architecture. In this project, our main task was to design and simulate a simple pipelined processor for feedforward linear threshold neural networks.

Figure 1 shows the layout of the neural network to be simulated. There are four input nodes (layer 1), four hidden nodes (layer 2) and four output nodes (layer 3). A binary input (0 or 1) is supplied to each input node. The hidden nodes have fixed weights of -1, 0 or +1. Similarly, the outputs are also binary values (0's and 1's).

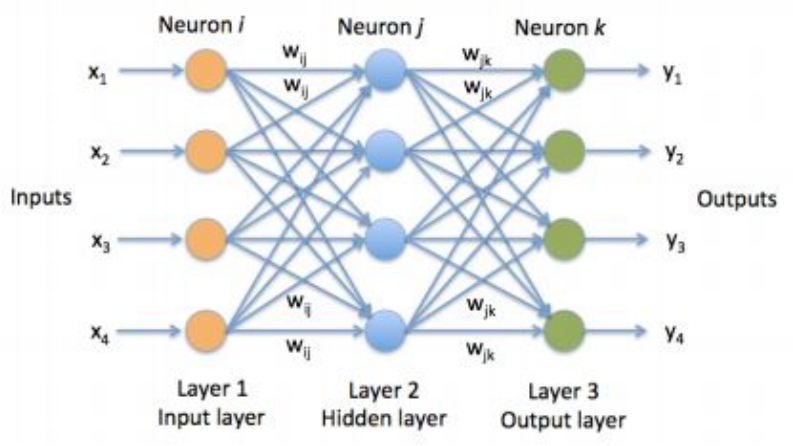

Figure 1. Neural network with 4 inputs, 4 hidden, and 4 output nodes. [8]

The hidden layer and output layer take in inputs from the input layer and hidden layer respectively and multiply with associated weights as shown in equation 1. If the result of the computation is negative, then zero is assigned to the result, whereas a value greater than or equal to zero gets a 1. Equation 2 depicts this calculation.

$$z = \sum_{i=1}^{4} x_i w_{ij} \qquad (1)$$

Equation 1. Each hidden neuron and output neuron sums up the inputs multiplied by the weights

$$y = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; if \; z < 0 \end{cases} \qquad (2)$$

Equation 2. The threshold function determines the output of each hidden neuron and output neuron

The design constraints for our project are as follows:

i. The processor architecture had to be pipelined.
ii. The processor had to be optimized for throughput.
iii. The processor had to be cycle accurate.
iv. All typical parts of the processor such as instruction decoder, ALU and registers had to be simulated.
v. The functional simulator had to capture the effect of running a simulated program on the simulated machine state.
vi. All the program instructions and data memory including inputs and hidden weights had to be in a single text file.
vii. The output of the simulation had to be published on a separate file or appended on the input file.

II. Design Approach

We have build a 3 stage pipeline processor to simulate the neural network in Verilog. Initially, we took motivation from the MIPS 5-stage pipeline processor. But we soon realized that we could simplify our architecture with just 3 stages (**Sec. III C.i**). For instance, unlike the MIPS 5-stage architecture, we don't need any hazard detection unit.

As shown in Figure 2, we have three stages in our pipeline indicating Fetch, Decode and Execute stages. We choose this three stage architecture because we wanted to have roughly the same amount of delay in each pipeline stage. However, the executing stage turned out to suffer the longest delay as it consists of two ALU's and a data memory. Although having a long executing stage limited our clock frequency, this wasn't a setback for our design. Because our design was evaluated on the basis of the clock-accuracy and number of clock cycles to complete the overall program rather than how fast we clock it, having a slightly longer execution stage didn't impede our project. Most importantly, having 3 pipeline stages were convenient for our design because we could get a clock-accurate functional neuromorphic architecture with no data and control hazards.

The following section describes each stage in our pipeline:

Fetch: This stage consists of a program counter and an instruction memory unit. In this stage, the processor reads from the instruction memory unit. At the end of this stage, the instruction is stored on the instruction fetch/decode pipeline register.

The program starts at PC 0 and in each clock cycle, PC increments by 1 until it reaches a halt instruction. We have a clock cycle counter that updates every time we fetch an instruction.

Decode: In this stage, the processor reads operands from the register file and decodes the instruction to generate the control signals. There are three modules in this stage. The register file contains the operands, the control unit decodes the control signals and the bit-field extend module extends the 16 bits input into 32 bit field filling all 16 most significant bits with zeros.

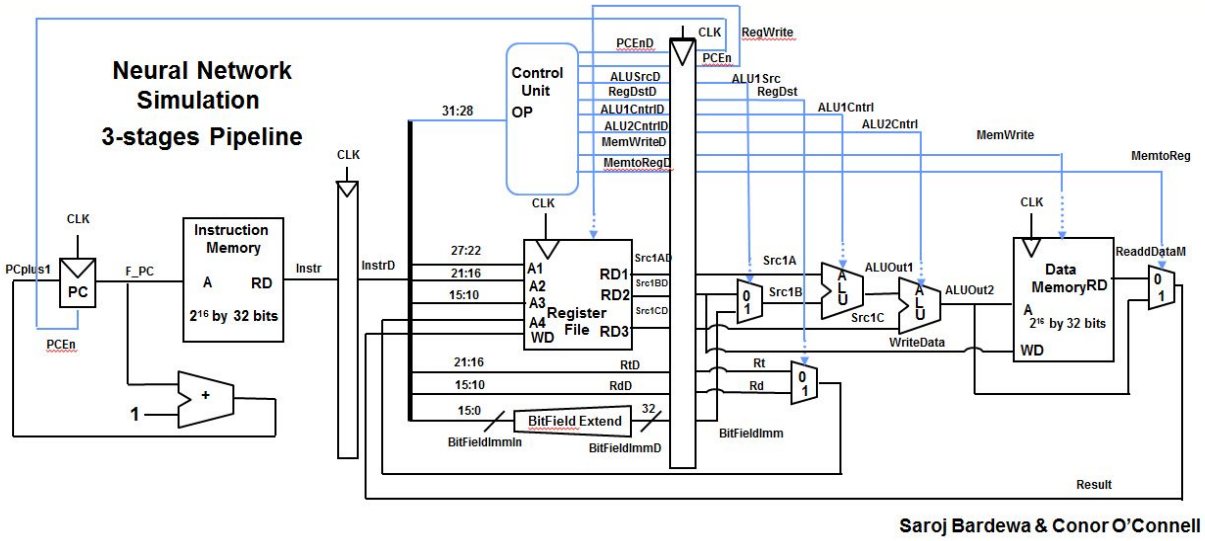

Figure 2. 3-stage pipeline layout

Execute:

The execute stage constitutes the longest stage in our 3-stage pipeline architecture. In this stage, the processor performs ALU operations, reads from or writes to data memory, and writes the result back to the register file. It consists of two ALUs, a data memory unit and three multiplexers. The second ALU is used specifically for a Multiply-and-Accumulate (MAC) operation (See Sec. III C.viii). Each multiplexer is used to make a specific selection:select the destination writeback register address, select the destination writeback register data, and select an ALU source.

Unlike a conventional 5-stage MIPS architecture that has separate execute, memory and writeback stages, our 3-stage pipeline architecture unifies all the functions of those stages into the execute unit. Going to a 3-stage design was advantageous to us because it eliminates data hazards. Also, we didn't require any extra hardware mechanisms to account for data forwarding or to regulate control hazards. Similarly, we eliminated a structural hazard by writing to the register file in the negative clock edge and reading from the register file on the positive clock edge. Because we were focused on processor clock cycle count rather than the clock speed, having a longer execution stage, despite slowing clock frequency, didn't impact our performance metrics.

III. Processor Architecture and Justifications

The following subsections provide more information on input/output memory, instruction format and design decisions.

A. Input/Output Memory

Our processor has separate instruction and data modules. This required us to be able to load two files for instruction and data. However, one of the design constraints was to be

able to load a single memory image file that contained both instruction and data memory. The way we handled this situation was by loading two copies of the same input file in our two separate instruction and data modules. In other words, we load the same file in both instruction memory and data memory.

The memory image contains 32-bit hex formatted instructions starting at offset 0. These are followed by the 32-bit hex network weights, with 1 weight stored per line, followed by the 32-bit hex input vectors. Each input in each vector is stored in a separate line. To read the data memory, we pre-compute the address where the data starts in our memory image. Then, at the first instruction of the image file, we load that address into R63. To load all other inputs and weights, we add corresponding offsets to R63.
Similarly, we store the output of the neural network to a separate file. This made it convenient for us to verify the expected output of the processor with the result from our verification program.

B. Instruction Format

Opcode (hex)	Mnemonic	Instruction	Opcode (hex)	Mnemonic	Instruction
0	NOP	No Operation	8	-	(Not specified)
1	ADD	Add	9	ADDI	Add Immediate
2	MUL	Multiply	A	J	Jump (not implemented)
3	SINN	Set If Not Negative	B	HALT	Halt computation
4	MAC	Multiply and Accumulate	C	-	(Not specified)
5	-	(Not specified)	D	-	(Not specified)
6	-	(Not specified)	E	LD	Load from memory
7	-	(Not specified)	F	ST	Store to memory

Table 1. Opcode encoding and mnemonic assignments

Our ISA takes inspiration from the MIPS architecture, but with numerous modifications described below. Like MIPS, it is a 32-bit RISC register-register (load-store) architecture. It shares MIPS's distinction between I-type (Immediate) and R-type (Register) instructions. Our R-type instruction leaves bits 0-9 unused, whereas MIPS uses the corresponding bits for an extension to the opcode field. We added instructions for Set-If-Not-Negative (SINN) and for Multiply-And-Accumulate (MAC) functions, because these were particularly helpful to our use-case. We split the overloaded MIPS ADD instruction into the I-type ADDI and the R-type ADD. As our algorithm is purely integer

based, we did not implement any of the floating point instructions present in the full MIPS instruction set.

Assembler

The Assembler takes an assembly program file as input, and generates an executable 32-bit hex-formatted machine code file usable by our CPU module. Because our CPU halts in the decode stage, the motif NOP - HALT - NOP is used to halt the processor. The NOP preceding the HALT prevents the processor from halting during the execution of the final operation of the algorithm. Although not strictly necessary, the NOP trailing the HALT prevents the processor from fetching data as if it were instructions.

Assembly Code Generator Script

The Assembly Code Generator script generates valid assembly code for the Assembler. It generates code that loads the neural network's weights, then generates code for each of the thousand test cases. This is where our loop unrolling happens. Finally, it generates code to halt the CPU.

Datafile Generator

The Datafile Generator randomly generates the network weights and the input vectors for the 1000 test cases, and simultaneously computes the expected results for those specific input vectors and prints them into a text file. This allows verification that the simulator is working as expected. The output of this code block must be combined with the output of the assembler to generate a useable memory image for the CPU module.

C. Design Justifications

i. 3 stage pipeline

We choose a three-stage pipeline architecture for our design. Initially, we also considered a 5-stage pipeline processor (shown in Figure 3). We did not consider more than five stages because there were not enough parallelizable stages we could exploit for our neural network optimized processor. Similarly, we eliminated the option of a five stage pipeline because the five stage design implementation we were considering suffered from data hazard, particularly raw hazard. However, the tradeoff of having only 3-stages in our pipeline was that the speedup of our pipeline was lower compared with more heavily pipelined processor. Also, the pipeline was not as well balanced: the execution stage was longer compared to the fetch and decode stages. Nevertheless, the benefit of eliminating data hazards and simplicity of writing assembly for the processor convinced us to take this option.

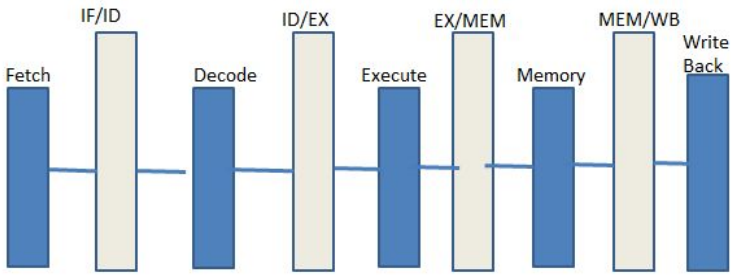

Figure 3. 5-stage pipeline layout

ii. Verilog for Implementing the design

We have used RTL level description using Verilog to design our processor. Our simulator captures the effect of running a simulated program on the simulated machine state. We have simulated each functional unit, so our design is synthesizable.

The tradeoff of using Verilog was that we had to do a lower level implementation, which would not have been necessary in a higher level language. However, we choose Verilog because it is a language we were both comfortable with. Additionally, we determined that it was simpler to implement clock synchronization and get global clock information using Verilog.

iii. R0 as constant zero

Following the lead of MIPS, we decided that R0 should be constant zero. This allows some conceptually different instructions to be implemented as a single opcode in the machine code. For example, we can implement load-from-absolute-address as load-from-base-plus-offset, with R0 as the base register. The tradeoff is that we lost R0 as a useable register, however we felt that this was was more than compensated for by what we gained, especially considering the number of register we were implementing: see section **Sec III C.iii**, below.

iv. 4 bit opcodes

We decided that our opcode should be 4 bits. At the time we made this decision, we only needed 6 different instructions, so we could have used 3 bits to store 8 values. However, two lines of reasoning lead us to use 4 bits: Firstly, our machine code is stored in hexadecimal format. If the opcode is 3 bits, then the next bit field would “leak” into the hex value; for example, E and F could both represent the same opcode. With 4 bits, the field break would be aligned with the hexadecimal digit break, creating machine code that is slightly more human readable. Secondly, 4 bits would allow us to define and assign more opcode values, should the need arise. This foresight proved useful later, when we added opcodes for NOP, MAC, ADDI and HALT. We also removed our jump/branch instruction, bringing our total to 9 defined opcodes. Had we needed to cut down to 8 instructions, we could have implemented our NOP instruction without a dedicated opcode, say by the instruction “ADD,R0,R0,R0”. However, it was cleaner to just use a dedicated opcode, and our early decision allowed us to do just that.

v. 64 registers

We decided to implement 64 registers in our ISA. This conferred the significant benefit of allowing us to permanently store the input-to-hidden and the hidden-to-output weights in registers, using only a single load instruction per weight. This saves us from repeatedly reloading weights each time they are needed, saving both execution time and instruction count. The tradeoff for this decision was that we needed 6 bits to specify a register. As we had 2 register fields in our I-type instruction format, this takes up 12 bits for specifying registers. Given that we had decided on 4 bit opcodes, this cut into our range of immediate values. However, we felt that the simplicity and speed gained by this approach was worth this loss of range.

vi. 32 bit instruction length

Having decided on 4 bits for our opcodes, and 6 bits for our register fields, we were left with the question: how long should our instructions be? This would have to be determined by the immediate value field. Immediate values are used in I-type instruction, which have 1 opcode field, 2 register fields, and the rest assigned to the immediate value field. It is traditional, though not strictly necessary, for instruction length to be a power of 2. This tradition arises because it makes interfacing with the memory system easier. In deference to this tradition, we looked at whether 32 bits would be sufficient for our needs. Our prior decisions had taken up 16 bits, leaving us with 16 bits for our immediate field. To know whether this was enough, we needed to know the largest value we would need to represent. This turned out to be the base address of the weights and data. Our code took 48 instructions per test case, and with 1000 test cases, that leads to 48000, plus a small constant term for loading the weights. Given that the max value that can be represented by 16 bits is 2^16 = 65535, we determined that a 16 bit immediate field was sufficient, assuming it was unsigned. Given that our algorithm never uses negative immediate values, we felt that this was an acceptable tradeoff.

vii. Word addressable memory

Our memory module is a word (32-bits) addressable memory. In other words, we have a non-byte addressable memory. This makes it easier for debugging purposes because we address 1 word at a time, and increment PC by 1 as well. So, it was easier to make a correlation between line number in the memory image data file, and instruction count in the processor simulation. Like our choice of 4-bit opcodes (section III C.iv), this design choice made our machine code more human readable. This design choice also reduced the number of addresses used for storing instructions. However, it increased the amount of memory that would be needed to physically implement our design. We felt that this was an acceptable tradeoff, because we were right at the limit of how many immediate value bits we needed, and increasing the number of addresses in the program would have pushed us over the edge.

viii. Loop unrolling / No Jump / No Branch instructions

In this assignment, there are no necessary jump or branch instructions, that is, it is possible to use loop unrolling over the entire algorithm, resulting in purely linear code. Doing this eliminates the need for the branch instruction, as it will never be used. However, there is a large memory size penalty to pay for doing this: each iteration over

the unrolled loop takes up its own space in memory. We decided that this was worth doing, despite the cost, because it eliminated control hazards, and also eliminated the branch instruction itself, saving 1 instruction per loop.

ix. Duplicated memory (for instruction and data)

As per our design constraint, we were supposed to use a single memory image that contained both instruction and data. Since, we have separate instruction and data memories, the way we resolve this issue is by loading same file in our data and instruction memories. The tradeoff is that we use twice the amount of memory for the same memory image. Nonetheless, this was an expedient solution to the structural hazard that would have occurred had we used a simple single data and instruction memory unit.

x. MAC instruction with two ALUs

We noticed that our algorithm was repeatedly multiplying two numbers, then adding the results together. Using additional hardware, it is possible to do more computation per instruction. We realized that we could speed this up by implementing the Multiply and Accumulate operation, which is common in digital signal processing applications. To do this, we would need an additional read port on our register file, with a corresponding address input, and an additional ALU in the execute stage. This would also make the stages of the pipeline less balanced, necessitating a slower clock speed. We decided that, given the significant improvements in code size and execution time in clock cycles, these were acceptable tradeoffs.

xi. ADDI instruction

In MIPS assembly, the ADD instruction is overloaded. In other words, whether the ADD instruction refers to adding two registers, or adding a register and an immediate value is determined by context. Conversely, we define ADDI as a new assembly mnemonic for adding a immediate value with a register, reserving ADD to only add a pair of registers. We did this to both make immediate instructions (I-type) explicit and distinct from R-type instructions, and to make writing the assembler easier.

III. Testing

We used Verilogger (Synaptic CAD) and ModelSim to compile and test our program. We have a global clock that is updated in every instruction. We wrote a companion program that generates random weights (-1, 0 or +1) and 1000 input vectors. The program also computes the expected output for the given set of inputs and weights, and writes them into a file. This expected output file is useful to compare the results from our processor.

We tested our program using an incremental methodology. First, we tested for a single input and observed its output. This was helped us to identify our bugs in the module. We could manually check the results at each stage, and debug the program. Once the architecture produced the correct result for a single output, we supplied 1000 input vectors and compared its outcome with the pre-calculated results from our companion program. The result generated by our pipeline processor was in agreement with the expected output.

IV. Performance

Our algorithm takes 48,034 instructions to simulate the 1000-vector 4-input, 4-hidden and 4-output neuron, plus the NOP - HALT - NOP ending motif of 3 instructions. At the start, it takes 3 clock cycles to fill the pipeline. After the pipeline is filled, we get one instruction completed per cycle.The most conspicuous part of our pipeline design is that we don't have any hazards, thus we never have to stall. Due to this reason, the number of clock cycles taken is equal to the number of instructions between the program start and the halt motif plus the number of cycles required to fill the pipeline. Thus, our processor take a total of 48037 clock cycles to complete.

V. Conclusion

Finally, we believe we have designed a processor that has multiple strong points in comparison to potential competing designs. Our simple, modular implementation is both maintainable and extendable. The design has been tuned precisely for the task at hand; hardware unnecessary for our application has been dropped, leading to a less expensive physical realization. The instructions in our instruction set have been chosen to optimise the execution time of the neural networks that the processor will be used to simulate. Similarly, because we know our main target is a 4 input, 4 hidden, and 4 output node network, we chose to give our processor more than enough registers to fit all 32 network weights of that network in memory, saving the program from having to load the weights more than once.

A major strength of our design is our complete mitigation of hazards. Our lack of jump or branch instructions guarantees that no control hazards can occur. Data hazards cannot happen in our 3 stage pipeline, because each result is computed and stored by the time the registers are read for the next instruction. Because we have separated our data and instruction memory, we don't have the structural hazard of two pipeline stages trying to access the same memory at the same time.

Our choice of Verilog as our design language confers several benefits. It is automatically clock accurate. It is more realistic than other designs in a higher level language. It can even be synthesized.

However, any design is a collection of tradeoffs, and ours is no different. Verilog may be clock accurate, but simulating all those components makes for a slow simulation. We repeatedly chose to use more memory to overcome hazards: we unrolled every loop in the program to avoid implementing branching and the control hazards that come with it, and we completely duplicated our memory to avoid a structural hazard. In a physical design, all this memory would increase the cost of our design. Our 3 stage pipeline avoids data hazards, but it also makes the critical stage longer, necessitating slower clocks.

We feel that our choices in these tradeoffs were well considered and reasonable. Our particular collection of design choices has lead to a processor that is optimised and well suited for the task it performs.

Bibliography

[1] Ebeling, Carl. “a Verilog description for an 4 x 16 register file” http://courses.cs.washington.edu/courses/cse370/10sp/pdfs/lectures/regfile.txt (06/01/2016)
[2] Tomczak, Czarek “How can I find script's directory with Python? [duplicate]” Stack Overflow http://stackoverflow.com/questions/4934806/how-can-i-find-scripts-directory-with-python (06/01/2016)
[3] Stack Overflow User #4134, “Marty”, “How to sign-extend a number in Verilog.” Stack Overflow http://stackoverflow.com/questions/4176556/how-to-sign-extend-a-number-in-verilog (06/01/2016)
[4] Stack Overflow User #465838, “Russ” “Open file in a relative location in Python.” Stack Overflow http://stackoverflow.com/questions/7165749/open-file-in-a-relative-location-in-python (06/01/2016)
[5] Harris, Sarah, and David Harris. Digital Design and Computer Architecture: ARM Edition. Morgan Kaufmann, 2015.
[6] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.
[7] Palnitkar, Samir. Verilog HDL: a guide to digital design and synthesis. Vol. 1. Prentice Hall Professional, 2003.
[8] Teuscher, Christof. “ece486_586_spring16_final_project_r1” https://d2l.pdx.edu/d2l/le/content/575286/viewContent/2463243/View?ou=575286 (06/01/2016)