Team Members: Saroj Bardewa and Conor O'Connell

Week #6

THIS WEEK:

* Designed Instruction Memory Unit
* Designed Data Memory Unit
* Designed Register File Unit
* Designed 2 and 4 input Multiplexer modules
* Designed Sign Extension Module
* Designed Control Unit
* Updated Assembler for 64 registers

**NEXT WEEK:**

* Work further on designing a basic pipeline NN processor.
* Extend capability of our simulation to load in a hex textfile and save the results to a hex file
* Complete the assembler and produce the instructions in hex format

**# Detail of this week:**

* Designed Instruction Memory Unit

The first step in pipeline is to read in instruction from the memory\_image. We designed an instruction memory unit that reads in and saves 32-bit of data in a memory bank. And the program can access the instruction in any order –it can process out of order execution.



***Instruction Memory Module Program***



Test bench for the instructionMemory Module

This program tests the InstructionMemory Module to verify the output of the module for given input combination. It is shown in figure 2.



**Result of the simulation:**

The result of the simulation gives the memory instruction accessed for a given PC value. The out of order execution makes the program very efficient. It is shown in figure 2.

**Assembler:**

Until now, we had been assuming that we would use 32 registers. This week, however, we decided that 64 registers would be better, because it would allow us to fit every input and neuron weight in the “base case” (the basic requirements, not including the extra credit) into a register. thus , we added support in assembler for up to 64 registers.

**Register File:**

The register file stores the contents of all 64 32-bit wide registers. As written now, it has 1 input ports and 2 output ports, but if we opt to implement the Multiply-and-Accumulate (MAC) function, it will have to be updated to have 3 output ports. During our ISA design, we decided that, like MIPS, R0 should be constant 0. This is the module where this functionality is implemented.

**Multiplexers:**

We designed 2 Multiplexer variants. One has 2 inputs and 1 select bit, and the other has 4 inputs and 2 select bits. Both variants take 32-bit wide inputs.

**Sign extension module:**

This week, we built a sign extension module. This module takes the immediate field, 10 bits wide in our implementation, and sign extends it to the 32 bits necessary for the ALU to use it in a calculation.

**Control Unit:**

The Control Unit we wrote this week generates the control signals necessary for the proper operation of the processor. This includes signals that control whether there is a write to a register, a write to memory, which operation the ALU should perform, and so on. If we opt to implement the Multiply-and-Accumulate (MAC) function, it will have to be updated to include control signals for the additional ALU.

**Test File:**

This test file contains the memory image of 32-bit address in hex format.



* **Designed Data memory Unit:**

Data memory unit is used to load and store data. The input values and internal weights are loaded into this module which is accessed for calculation. Also, the output of the neural network is stored in this unit, and which is further written into a file.



*Data Memory Module:*

*TestBench of Data Memory module:*



*Output of the module:*

*Test Input File:*



*Test Output File Generated from the module:*

