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
- Designed the Neural Network Simulator Architecture

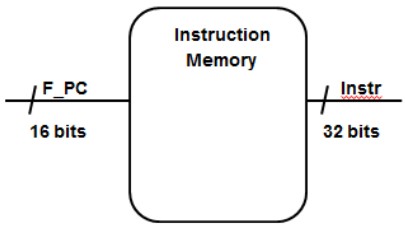
NEXT WEEK:

- Debug the code
- Extend the capability to support more than 4 input neurons

 # 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

```
/* PURPOSE: This is the module that contains memory image in hex formate
 *          The memory can be accessed depending on the PC value
 * INPUT    : PC address
 * OUTPUT   : Instruction at the memory location
 * Saroj Bardewa
 * ECE 586
 */

module instructionMemory(F_PC,Instr);
    parameter OUT_BUS_WIDTH=32;             // Output Bus Width
    parameter IN_BUS_WIDTH=16;              // Input Bus Width
    parameter MEMORY_WIDTH=32;              // Bits of Memory accessed at a time
    parameter ADDRESS_SIZE=2**IN_BUS_WIDTH; // Size of memory bank. Depends on locations
                                            // that can be referenced by a Program Counter

    input[IN_BUS_WIDTH-1:0] F_PC;           //input PC
    output reg [OUT_BUS_WIDTH-1:0] Instr;               // Hex memory

    reg[MEMORY_WIDTH-1:0] instrctMemoryBank[ADDRESS_SIZE-1:0]; //32 bits 2^16 registers

    initial
        begin
            $readmemh("memoryImage.txt",instrctMemoryBank);  //Read Memory Image
        end

    always@(F_PC)
        begin
            Instr = instrctMemoryBank[F_PC];  // Read F_PC = 0 --> first eight bits
        end
endmodule
```

Test bench for the instructionMemory Module

This program tests the InstructionMemory Module to verify the output of the module for given input combination.

```
/* PURPOSE: This is the test bed for instructMemory Module
 * INPUT: None
 * OUTPUT: None

 * Saroj Bardewa
 */

module instructMem_Test();
parameter OUT_BUS_WIDTH=32;             //Output Bus Width
parameter IN_BUS_WIDTH =16;             //Input Bus Width

reg[IN_BUS_WIDTH-1:0] F_PC;          //input PC
wire[OUT_BUS_WIDTH-1:0] Instr;               // Hex memory

instructionMemory IM_1(F_PC,Instr);

initial
    begin
        F_PC = 0;   // zeroth address
    #10 F_PC = 1;   // first address
    #10 F_PC = 2;
    #10 F_PC = 3;
    #10 F_PC = 6;   //Out of Order
    #20 F_PC = 1;   //Out of order
    #20 F_PC = 9;

    #20 $finish();
    end
endmodule
```

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.

Test File:

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

```
21310000
21310001
21310002
21310003
21310004
21310005
21310006
21310007
21310008
21310009
```

- **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.

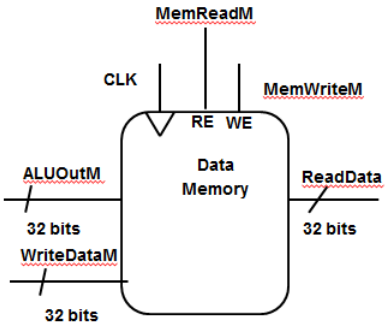


Figure: Data Memory Unit

Data Memory Module:

```
/*******************************
 * PURPOSE: Load and store of data
 *          takes place on Data Memory
 *          It is used for I-type of instruction.
 *  INPUTS: Clock, Memory Write Enable, ALU Memory Address,
 *          Write Data
 *  OUTPUT: Stored Data
 *  AUTHOR: Saroj Bardewa
 *******************************/

module dataMemory(CLK,writeEn,readEn,ALUMemAdd,writeDataM,readDataW);
   parameter DATA_BASE_ADD = 5;        // Starting data base address
   parameter OUTPUT_FILE_SIZE = 4;         // Depends on the number of output to write
   parameter IN_BUS_WIDTH=32;
   parameter MEMORY_WIDTH=32;              // Bits of Memory accessed at a time
   parameter ADDRESS_SIZE=20;              // Size of memory bank
                                          // that can be referenced by a Address size
   integer file;
   input   CLK, writeEn,readEn;
   input   [IN_BUS_WIDTH-1:0] ALUMemAdd;
   input   [MEMORY_WIDTH-1:0] writeDataM;  // 32bit data value
   output reg signed [MEMORY_WIDTH-1:0] readDataW;


   /* At positive clock edge, if there is write enable, the module latches in the data
    * specified by the ALU memory address. If it is load, then the module reads out the
    * value stored at the particular register specified by the input address */

   reg signed [MEMORY_WIDTH-1:0] dataMemoryBank[0:ADDRESS_SIZE-1]; //#of locations = ADDRESS_SIZE each with MEMORY_WIDTH size
   reg signed [MEMORY_WIDTH-1:0] outputMemoryBank[0:OUTPUT_FILE_SIZE];

   initial $readmemh("dataMemoryFile.txt",dataMemoryBank);  //Read Memory Image
   initial file = $fopen("output.txt","w");    // Initially the file is empty

   //Write Data is clock synchronous
   always@(posedge CLK,ALUMemAdd,writeEn,writeDataM,readEn)
       begin
           if(writeEn)
               begin
                   outputMemoryBank[ALUMemAdd] = writeDataM ;  // Read F_PC = 0 --> first eight bits
                   $fdisplay(file,outputMemoryBank[ALUMemAdd]);  // Write the value to the file
               end
           else if(readEn)
               begin
                   readDataW = dataMemoryBank[ALUMemAdd+DATA_BASE_ADD];  //Read from an address and output the data
                   $display("Read %d from the Address: %d",readDataW,ALUMemAdd);
               end
           else
               $display("No Memory Access!");
           end

endmodule
```

Test bench of Data module:

```
/* PURPOSE: This is the test bench for dataMemory Module
 * INPUT: None
 * OUTPUT: None
 * Saroj Bardewa
 */
module dataMem_Test();
    parameter IN_BUS_WIDTH=32;
    parameter MEMORY_WIDTH=32;              // Bits of Memory accessed at a time

    reg     CLK, writeEn,readEn;
    reg     [IN_BUS_WIDTH-1:0] ALUMemAdd;
    reg     [MEMORY_WIDTH-1:0] writeDataM;  // 32bit data value
    wire    [MEMORY_WIDTH-1:0] readDataW;

initial
    begin
        CLK = 0;
    end

//Generate Clock
always
    begin
        CLK = ~CLK;
        #5;
    end

dataMemory Data1 (CLK,writeEn,readEn,ALUMemAdd,writeDataM,readDataW);

initial
    begin
        writeEn = 0; readEn = 1; ALUMemAdd = 0; writeDataM = 0;  // zeroth address
    #10 writeEn = 0; readEn = 1; ALUMemAdd = 1; writeDataM = 0;
    #10 writeEn = 0; readEn = 1; ALUMemAdd = 2; writeDataM = 0;
    #10 writeEn = 0; readEn = 1; ALUMemAdd = 3; writeDataM = 0;
    #10 writeEn = 0; readEn = 1; ALUMemAdd = 4; writeDataM = 0;
    #10 writeEn = 0; readEn = 1; ALUMemAdd = 5; writeDataM = 0;
    #10 writeEn = 1; readEn = 0; ALUMemAdd = 0; writeDataM = 1;   //Write to a file
    #10 writeEn = 1; readEn = 0; ALUMemAdd = 1; writeDataM = 1;  // Write to a file
    #10 writeEn = 1; readEn = 0; ALUMemAdd = 2; writeDataM = 0;   // Write to a file
    #10 writeEn = 1; readEn = 0; ALUMemAdd = 2; writeDataM = -1;   // Write to a file

    #10 $finish();

    end

endmodule
```

Test Input File:

```
10101010
20101011
30101013
00000000
00000000
00000001
00000000
ffffffff
00000001
00000000
ffffffff
```

Test Output File Generated from the module:

```
 1
 1
 0
-1
```

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.

Our 3-Stage Pipeline Architecture: