# **Verilog Portfolio Documentation**

## Project 1: Neural Network Processor

### Component Descriptions:

#### Register File

The main purpose of the register file is to hold, and dynamically change the register values on processor command. It contains a number of inputs:

* **clk:** This is the clock block providing synchronous functioning to the RegisterFile module. It is used to update registers synchronously as in this case.
* **read\_addr1, read\_addr2:** these are 5-bit elements that signify which register the read\_addr can read from. Using 5 bits, one can represent 32 unique addresses, allowing to add a 32-bit input into a register.
* write\_addr: the write-adder can do the same for the write-date object that has 32 bits, allowing it to register.
* write\_data: this is the updated data that we want to register.
* reg\_write: this is the control signal that allows the update procedure. It is either high(1) or low(0), and depending on that, it goes through with the write\_data update into the register.

If there is any change within the read\_addrs, then the @(\*) condition is triggered, and the register values within the binary coordinates determined by the read\_addrs are supplied to the outputs read\_data1, and read\_data2.

| module RegisterFile(  input clk,  input [4:0] read\_addr1, read\_addr2, write\_addr,  input [31:0] write\_data,  input reg\_write,  output reg [31:0] read\_data1, read\_data2 );  reg [31:0] registers [0:31];   always @(posedge clk) begin  if (reg\_write) begin  registers[write\_addr] <= write\_data;  end  end   always @(\*) begin  read\_data1 = registers[read\_addr1];  read\_data2 = registers[read\_addr2];  end endmodule |
| --- |

#### Instruction Memory

The main purpose of this module is to hold the instruction codes that allow the processor to switch its state. This is representative of a Finite State Machine. The module declaration takes in an addr input that can represent 128 unique addresses. It also creates an output register that is not exactly the same as a register but can hold values such as the instruction for multiple clock cycles. Here we have three states:

* Null State: This does nothing.
* Loading State: In this state, the write\_data value is updated.
* Gradient Descent Calculation Step: The processor calculates the gradient descent in this state.

Whenever the addr is changed, it adds the memory that is determined with the addr to the instruction.

| module InstructionMemory(  input [7:0] addr,  output reg [31:0] instruction );  reg [31:0] memory [0:255];   initial begin    memory[0] = 32'h00000000; // Null  memory[1] = 32'h00000001; // Loading State  memory[2] = 32'h00000002; // Gradient Descent Calculation Step    end   always @(\*) begin  instruction = memory[addr];  end endmodule |
| --- |

#### 

#### 

#### Program Counter

This module provides synchronicity, and update procedures that are related to the timing to the program. The clock and reset inputs provide a functionality where if the postedge (rising edge of a specified clock signal) reset is high (1), then the pc is registered as 0. If not, then the next\_addr input is registered to pc.

| module ProgramCounter(  input clk,  input reset,  input [7:0] next\_addr,  output reg [7:0] pc );  always @(posedge clk or posedge reset) begin  if (reset) begin  pc <= 0;   end else begin  pc <= next\_addr;  end  end endmodule |
| --- |

#### 

#### 

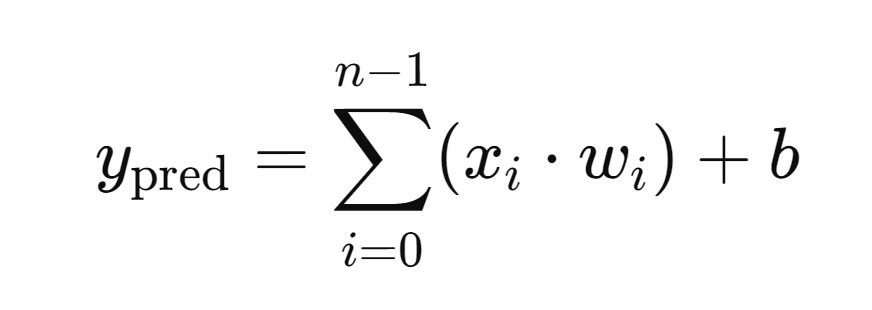
#### Control Unit

The Control Unit is responsible for getting the instructions from the memory, and based on what instruction is handled at that time, process it, and move on to the next one. Here, the last 8-bits are used to determine the class of the instruction (0, 1, or 2). Based on the result, case statement employs the correct instruction and reveals the necessary values for the completion of that instruction, such as in the case of 01, it first makes the reg\_write high, so that the if statement goes through with the update procedures in the register file. It then determines the write\_addr (the address provider for write\_data) (next 5 bits from right to left), and the read\_addr1(1 being the instruction code it is updated under). It then outputs the next\_pc variable as +1, so that the next procedure takes place.

| module ControlUnit(  input [31:0] instruction,  output reg reg\_write,  output reg [4:0] write\_addr,  output reg [4:0] read\_addr1, read\_addr2,  output reg [7:0] next\_pc );  always @(\*) begin    case (instruction[31:24])  8'h00: begin // NOP  reg\_write = 0;  next\_pc = 1;  end  8'h01: begin  reg\_write = 1;  write\_addr = instruction[23:19];  read\_addr1 = instruction[18:14];  next\_pc = 2;  end  8'h02: begin  reg\_write = 0;  next\_pc = 3;  end  default: begin  reg\_write = 0;  next\_pc = 0;   end  endcase  end endmodule |
| --- |

### Algorithm Summary:

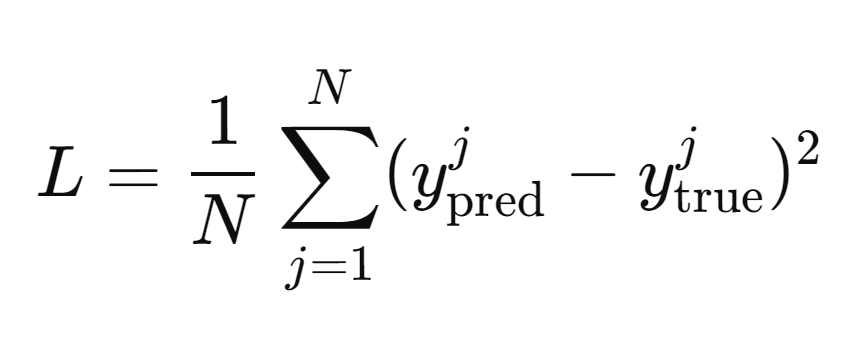
The neural network algorithm functions by iteratively adjusting weights and biases to minimize the prediction error on training data. The process begins with the forward propagation step, where the predicted output ypred​ is computed using the equation:



In our code this is done by:

| y\_pred = 0;  (i = 0; i < WIDTH; i = i + 1) begin y\_pred = y\_pred + (x[i] \* weights[i]);  y\_pred = y\_pred + bias; |
| --- |

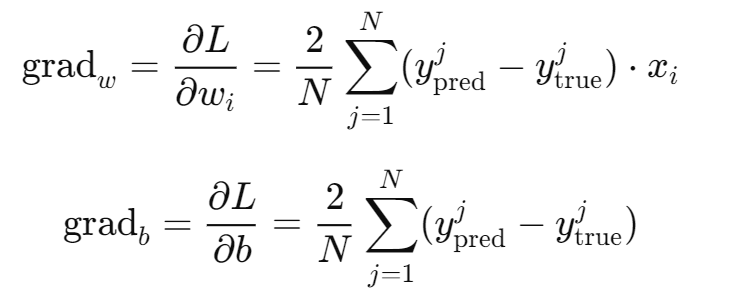
Next, the loss(L) is calculated to quantify the prediction error, typically using mean squared error (MSE):



In the code, this is captured by the following line:

| loss = (y\_pred - y\_true[0]) \* (y\_pred - y\_true[0]); |
| --- |

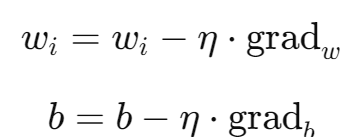
To update the weights and bias, gradients are computed through backpropagation, yielding the following equations:



These calculations are implemented in the code as follows:

| for (i = 0; i < WIDTH; i = i + 1) begin  grad\_w[i] = 2 \* (y\_pred - y\_true[0]) \* x[i]; end grad\_b = 2 \* (y\_pred - y\_true[0]); |
| --- |

Finally, these gradients are used to update the weights and bias using gradient descent with a specified learning rate η\etaη:



In the code, this update process is represented by:

| for (i = 0; i < WIDTH; i = i + 1) begin  weights[i] = weights[i] - (LEARNING\_RATE \* grad\_w[i]); end bias = bias - (LEARNING\_RATE \* grad\_b); |
| --- |

By repeating these steps over multiple epochs and iterating through the training data, the algorithm effectively minimizes the loss function, leading to improved accuracy in predicting outputs on unseen data.

#### Gradient Descent Neural Network Processor

This module works as the top module, calling the instances from all other modules, and also the module where the algorithm takes place. We first define a number of parameters necessary for our calculations, such as the number of samples, learning rate, and width. After defining the parameters. We then instantiate the components from our modules. The algorithm goes through and the weights and predicted values are added to the write\_data.

| module NeuralNetworkProcessor(  input clk,  input reset );    parameter WIDTH = 10;  parameter LEARNING\_RATE = 32'h00000001;  parameter NUM\_SAMPLES = 10;     reg [31:0] x [0:WIDTH-1];  reg [31:0] y\_true [0:NUM\_SAMPLES-1];  reg [31:0] weights [0:WIDTH-1];  reg [31:0] bias;  reg [31:0] y\_pred;  reg [31:0] grad\_w [0:WIDTH-1];   reg [31:0] grad\_b;  reg [31:0] loss;   reg [31:0] sum\_loss; // Accumulated loss for averaging   // Instruction memory interface  wire [31:0] instruction;  wire [7:0] pc\_out;  wire reg\_write;  wire [4:0] write\_addr, read\_addr1, read\_addr2;  wire [31:0] read\_data1, read\_data2, write\_data; // Register data   // Control signals  reg [7:0] next\_pc;  reg start\_training;  integer i; // Loop index   // Instantiate components  ProgramCounter pc(  .clk(clk),  .reset(reset),  .next\_addr(next\_pc),  .pc(pc\_out)  );   InstructionMemory imem(  .addr(pc\_out),  .instruction(instruction)  );   ControlUnit control(  .instruction(instruction),  .reg\_write(reg\_write),  .write\_addr(write\_addr),  .read\_addr1(read\_addr1),  .read\_addr2(read\_addr2),  .next\_pc(next\_pc)  );   RegisterFile reg\_file(  .clk(clk),  .read\_addr1(read\_addr1),  .read\_addr2(read\_addr2),  .write\_addr(write\_addr),  .write\_data(write\_data),  .reg\_write(reg\_write),  .read\_data1(read\_data1),  .read\_data2(read\_data2)  );     initial begin    x[0] = 32'h00000001;  x[1] = 32'h00000002;    y\_true[0] = 32'h00000003;  y\_true[1] = 32'h00000006;   for (i = 0; i < WIDTH; i = i + 1) begin  weights[i] = 32'h00000001;  end  bias = 32'h00000001;   sum\_loss = 0;  end  always @(posedge clk or posedge reset) begin  if (reset) begin  sum\_loss <= 0;  end else if (start\_training) begin    y\_pred = 0;  for (i = 0; i < WIDTH; i = i + 1) begin  y\_pred = y\_pred + (x[i] \* weights[i]);  end  y\_pred = y\_pred + bias;  loss = (y\_pred - y\_true[0]) \* (y\_pred - y\_true[0]);  sum\_loss = sum\_loss + loss;   for (i = 0; i < WIDTH; i = i + 1) begin  grad\_w[i] = 2 \* (y\_pred - y\_true[0]) \* x[i];  end  grad\_b = 2 \* (y\_pred - y\_true[0]);     for (i = 0; i < WIDTH; i = i + 1) begin  weights[i] = weights[i] - (LEARNING\_RATE \* grad\_w[i]);  end  bias = bias - (LEARNING\_RATE \* grad\_b);     write\_data = y\_pred;  if (reg\_write) begin  reg\_file.write\_data = write\_data;  end      for (i = 0; i < WIDTH; i = i + 1) begin  write\_addr = i;  write\_data = weights[i];  if (reg\_write) begin  reg\_file.write\_data = write\_data;  end  end  end  end   endmodule |
| --- |

**Project 2: SMP Cache Coherency Protocol**

1. **Processor Interface Module**

This module forwards requests from the processor to the cache memory and gets data. Inputs clk and reset are used for clock and reset. Variable clk ensures all operations are synchronized to the same clock signal. Reset initializes or resists the module to a known state, which is important for the correct operation when starting the system.

The process signals are proc\_addr, proc\_rw, proc\_data\_in, proc\_data\_out. In pro\_addr the processor sends the signal to specify the address in memory it wants to access. proc\_rw is used to indicate whether the operation is read or written, 0 or 1. proc\_data\_in is where data to be written into memory proc\_rw is set to 1. proc\_data\_out is similar to proc\_data\_in, the only difference is it is used to return data to the processor when proc\_rw is set to 0.

Cache Signals:

* cache\_rw, cache\_addr, cache\_data\_in, cache\_data\_out, cache\_hit
* cache\_rw is the signal to the cache to perform either read or write passed from the processor’s proc\_rw
* cache\_addr is the address sent to the cache, directly mapped from the processor’s proc\_addr
* cache\_data\_in sends data to the cache from the processor on a write operation:
* cache\_data\_out is the data returned from the cache when it reads data from the cache
* cache\_hit indicates whether the requested address is found in the cache and is valid.

The statement inside and always block is executed sequentially and function as a trigger; in this case, triggers the bock on the rising edge of the clock (clk) or when the reset signal goes high.

| module processor\_interface (  input clk,  input reset,  input [31:0] proc\_addr,   input proc\_rw,   input [31:0] proc\_data\_in,   output reg [31:0] proc\_data\_out,  output reg cache\_rw,   output reg [31:0] cache\_addr,   output reg [31:0] cache\_data\_in,   input [31:0] cache\_data\_out,   input cache\_hit  );  always @(posedge clk or posedge reset) begin  if (reset) begin  proc\_data\_out <= 32'b0;  cache\_rw <= 1'b0;  cache\_addr <= 32'b0;  cache\_data\_in <= 32'b0;  end  else begin  cache\_addr <= proc\_addr;  cache\_rw <= proc\_rw;  if (proc\_rw) begin  cache\_data\_in <= proc\_data\_in;  end else if (cache\_hit) begin  proc\_data\_out <= cache\_data\_out;   end  end end  endmodule |
| --- |

1. **Cache Controller Module**

Inputs:

* 1. clk: The clock signal that synchronizes operations.
  2. reset: Resets the cache controller to its initial state.
  3. addr: The memory address being accessed.
  4. rw: Read/Write control signal from the processor .
  5. data\_in: Data to be written to the cache from the processor.
  6. bus\_grant: Indicates that the bus has been granted to this cache.
  7. bus\_data\_in: Data received from the bus when accessing memory.
  8. bus\_rw: Bus Read/Write control signal.
  9. snoop\_hit: Indicates if another cache has the requested data.

Outputs:

* 1. data\_out: Data read from the cache to be sent to the processor.
  2. cache\_hit: Indicates whether the requested address is present in the cache.
  3. bus\_request: Signals that the cache needs access to the bus.
  4. bus\_data\_out: Data sent out on the bus during a write operation.
  5. invalidate: Signal to invalidate cache lines when necessary.

Internal Registers:

* 1. cache\_state: Stores the cache's current state which are INVLAID, SHARED, and MODIFIED.
  2. cache\_data: The cached data for the current address.
  3. cache\_tag: The tag portion of the cache to track stored addresses.

The statement inside and the always block triggers the block to execute on the rising edge of the clock clk or when the reset signal goes to high, ensuring synchronous updates with the clock or resetting the state when required.

| module cache\_controller (  input clk,  input reset,  input [31:0] addr,  input rw,  input [31:0] data\_in,  output reg [31:0] data\_out,  output reg cache\_hit,  output reg bus\_request,  input bus\_grant,  input [31:0] bus\_data\_in,  output reg [31:0] bus\_data\_out,  input bus\_rw,  output reg invalidate,  input snoop\_hit );  localparam INVALID = 2'b00,  SHARED = 2'b01,  MODIFIED = 2'b10;  reg [1:0] cache\_state; reg [31:0] cache\_data; reg [31:0] cache\_tag;  always @(posedge clk or posedge reset) begin  if (reset) begin  cache\_state <= INVALID;  cache\_tag <= 32'b0;  bus\_request <= 0;  invalidate <= 0;  end else begin  case (cache\_state)  INVALID: begin  if (rw) begin  bus\_request <= 1;  bus\_rw <= 1;  bus\_data\_out <= data\_in;  end else begin  bus\_request <= 1;  bus\_rw <= 0;  end  cache\_hit <= 0;  end    SHARED: begin  if (rw) begin  bus\_request <= 1;  bus\_rw <= 1;  bus\_data\_out <= data\_in;  cache\_state <= MODIFIED;  end else begin  data\_out <= cache\_data;  cache\_hit <= 1;  end  end    MODIFIED: begin  if (rw) begin  cache\_data <= data\_in;  end else begin  data\_out <= cache\_data;  cache\_hit <= 1;  end  if (bus\_grant) begin  bus\_request <= 1;  bus\_rw <= 1;  bus\_data\_out <= cache\_data;  end  end  endcase    if (snoop\_hit) begin  invalidate <= 1;  end else begin  invalidate <= 0;  end    if (bus\_grant) begin  if (rw) begin  cache\_data <= bus\_data\_in;  cache\_tag <= addr[31:4];  cache\_state <= MODIFIED;  end else begin  data\_out <= bus\_data\_in;  cache\_hit <= 1;  end  end  end end  endmodule |
| --- |

1. **Bus Interface Module**

### Inputs:

* clk: The clock signal that synchronizes operations.
* reset: Resets the bus interface to its initial state.
* bus\_request: Signal indicating a request for access to the bus.
* bus\_data\_in: Data received from the bus during read operations.
* cache\_hit: Indicates whether the requested address exists in the cache.

### Outputs:

* bus\_grant: Signal that grants access to the bus, allowing the cache to communicate over it.
* bus\_data\_out: Data to be sent out on the bus during write operations.
* bus\_rw: Indicates whether the operation is a read or write.
* invalidate: Signal to invalidate cache lines when necessary.

| module bus\_interface (  input clk,  input reset,  input bus\_request,  output reg bus\_grant,  input [31:0] bus\_data\_in,  output reg [31:0] bus\_data\_out,  output reg bus\_rw,  input cache\_hit,  output reg invalidate );  always @(posedge clk or posedge reset) begin  if (reset) begin  bus\_grant <= 0;  bus\_data\_out <= 32'b0;  bus\_rw <= 0;  invalidate <= 0;  end else begin  if (bus\_request) begin  bus\_grant <= 1;  bus\_rw <= 1;   bus\_data\_out <= bus\_data\_in;  invalidate <= 0;   end else if (cache\_hit) begin  bus\_grant <= 1;  bus\_rw <= 0;   bus\_data\_out <= 32'b0;   invalidate <= 1;   end else begin  bus\_grant <= 0;  end  end end  endmodule |
| --- |

1. **Cache Memory Module**

### Inputs:

* clk: The clock signal that synchronizes operations.
* reset: Resets the cache memory to its initial state.
* addr: The memory address for accessing cache data.
* rw: Read/Write control signal low value is read operation and high value is write operation.
* data\_in: Data to be written to the cache when a write operation occurs.
* invalidate: Signal to invalidate specific cache lines.
* cache\_hit: Indicates whether the requested address is found in the cache.

### Outputs:

* data\_out: Data read from the cache, provided to the processor on a read operation.

### Internal Registers:

* cache\_data [0:15]: An array that stores the cached data for 16 cache lines.
* cache\_tags [0:15]: An array that holds the tags associated with each cache line, used for address matching.
* cache\_states [0:15]: An array that tracks the state of each cache line (e.g., invalid, shared, modified).

| module cache\_memory (  input clk,  input reset,  input [31:0] addr,  input rw,  input [31:0] data\_in,  output reg [31:0] data\_out,  input invalidate,  input cache\_hit );  reg [31:0] cache\_data [0:15];  reg [31:0] cache\_tags [0:15];  reg [1:0] cache\_states [0:15];   always @(posedge clk or posedge reset) begin  if (reset) begin  integer i;  for (i = 0; i < 16; i = i + 1) begin  cache\_data[i] <= 32'b0;  cache\_tags[i] <= 32'b0;  cache\_states[i] <= 2'b00;  end  end else begin  integer index;  index = addr[3:0];   if (invalidate) begin  cache\_states[index] <= 2'b00;  end else if (rw) begin  cache\_data[index] <= data\_in;  cache\_tags[index] <= addr[31:4];  cache\_states[index] <= 2'b10;  end else begin  if (cache\_states[index] == 2'b10 || cache\_states[index] == 2'b01) begin  data\_out <= cache\_data[index];  end else begin  data\_out <= 32'b0;  end  end  end end  endmodule |
| --- |

1. **Top-Level Module**

### Inputs:

* clk: Clock signal to synchronize all modules.
* reset: Resets the system.
* addr: Memory address for cache access.
* rw: Read/Write control signal, low value for read and high value for write.
* data\_in: Data to be written to the cache.
* bus\_grant: Indicates if the bus has been granted to this module.
* bus\_data\_in: Data coming from the bus during a read operation.
* bus\_rw: Read/Write signal from the bus.
* snoop\_hit: Signal indicating another processor's access to a cache line.

### Outputs:

* data\_out: Data being sent out of the cache during a read.
* cache\_hit: Indicates if the requested address is present in the cache.
* bus\_request: Request signal for bus access.
* bus\_data\_out: Data to be sent over the bus.
* invalidate: Signal to invalidate a cache line.

| module top\_level (  input clk,  input reset,  input [31:0] addr,  input rw,  input [31:0] data\_in,  output [31:0] data\_out,  output cache\_hit,  output bus\_request,  input bus\_grant,  input [31:0] bus\_data\_in,  output [31:0] bus\_data\_out,  input bus\_rw,  output invalidate,  input snoop\_hit );  wire [31:0] cache\_data\_out; wire [31:0] cache\_memory\_data\_out;  cache\_memory cache (  .clk(clk),  .reset(reset),  .addr(addr),  .rw(rw),  .data\_in(data\_in),  .data\_out(cache\_data\_out),  .invalidate(invalidate),  .cache\_hit(cache\_hit) );  bus\_interface bus (  .clk(clk),  .reset(reset),  .bus\_request(bus\_request),  .bus\_grant(bus\_grant),  .bus\_data\_in(bus\_data\_in),  .bus\_data\_out(bus\_data\_out),  .bus\_rw(bus\_rw),  .cache\_hit(cache\_hit),  .invalidate(invalidate) );  cache\_controller controller (  .clk(clk),  .reset(reset),  .addr(addr),  .rw(rw),  .data\_in(data\_in),  .data\_out(data\_out),  .cache\_hit(cache\_hit),  .bus\_request(bus\_request),  .bus\_grant(bus\_grant),  .bus\_data\_in(bus\_data\_in),  .bus\_data\_out(bus\_data\_out),  .bus\_rw(bus\_rw),  .invalidate(invalidate),  .snoop\_hit(snoop\_hit) );  endmodule |
| --- |