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 y_{pred} is computed using the equation:

$$y_{\text{pred}} = \sum_{i=0}^{n-1} (x_i \cdot w_i) + b$$

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):

$$L = \frac{1}{N} \sum_{j=1}^{N} (y_{\text{pred}}^{j} - y_{\text{true}}^{j})^2$$

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:

$$\text{grad}_w = \frac{\partial L}{\partial w_i} = \frac{2}{N} \sum_{j=1}^{N} (y_{\text{pred}}^{j} - y_{\text{true}}^{j}) \cdot x_i$$

$$\text{grad}_b = \frac{\partial L}{\partial b} = \frac{2}{N} \sum_{j=1}^{N} (y_{\text{pred}}^{j} - y_{\text{true}}^{j})$$

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:

$$w_i = w_i - \eta \cdot \text{grad}_w$$

$$b = b - \eta \cdot \text{grad}_b$$

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
```

2. **Cache Controller Module**

   Inputs:

   a. clk: The clock signal that synchronizes operations.
   b. reset: Resets the cache controller to its initial state.
   c. addr: The memory address being accessed.
   d. rw: Read/Write control signal from the processor .
   e. data_in: Data to be written to the cache from the processor.
   f. bus_grant: Indicates that the bus has been granted to this cache.
   g. bus_data_in: Data received from the bus when accessing memory.
   h. bus_rw: Bus Read/Write control signal.
   i. snoop_hit: Indicates if another cache has the requested data.

   Outputs:

   j. data_out: Data read from the cache to be sent to the processor.
   k. cache_hit: Indicates whether the requested address is present in the cache.
   l. bus_request: Signals that the cache needs access to the bus.
   m. bus_data_out: Data sent out on the bus during a write operation.
   n. invalidate: Signal to invalidate cache lines when necessary.

   Internal Registers:

   o. cache_state: Stores the cache's current state which are INVLAID, SHARED, and MODIFIED.
   p. cache_data: The cached data for the current address.
   q. 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
```

3. 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
```

4. **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
```

5. **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
```