LLM-assisted Chip Design(Challenge-4)

Learning Goals

- Experiment with LLM-assisted chip design.
- Do “vibe coding” and experience the problems associated with it

1. Replicate the Johns Hopkins paper

I (means ChatGPT here) replicated the paper: *Designing Silicon Brains using LLM: Leveraging ChatGPT for Automated Description of a Spiking Neuron Array* ([arXiv:2402.10920](https://arxiv.org/abs/2402.10920)). The work involved:

- Designing a Leaky Integrate-and-Fire (LIF) neuron in Verilog using ChatGPT.
- Building a 2-layer spiking neural network.
- Creating a programmable SPI interface
- Integrating all modules in a top-level design

The LLM-generated code needed debugging and prompt iterations, which helped refine both design logic and synthesis readiness.

2. Use your favorite LLM

I used ChatGPT-4 to generate and refine the hardware modules. It was particularly effective for producing boilerplate Verilog code and made quick iterations possible with prompt adjustments.

3. Experiment with the queries and see what happens

I started with generic prompts like:

- Write a Verilog module for a LIF neuron.
- Make the parameters programmable.
- Create a SPI module to update parameters.
- Integrate a neuron network with weights'

Issues such as syntax errors, SystemVerilog usage, and multiply-driven nets led to refined prompts like:

- Replace enum with localparams.
- Use only Verilog constructs.
- Fix wire/reg conflicts.
- Flatten 2D arrays for port compatibility.

4. Keep track of all the queries you made

Here’s a summary of the types of queries and fixes:

- Declaring 'spike' as reg instead of wire.
- Handling membrane potential overflow and underflow.
- Flattening 2D ports for Verilog compatibility.

- Adding generate loops for scalable neuron instantiation.
- Making the parameters externally programmable.
- Separating combinational and sequential logic correctly.
- Resolving multiply driven signals in SPI module

6. Compare the results of your version with their paper

My version reflected the same architecture and approach used in the paper. Similar bugs appeared:

- Unpacked arrays in ports.
- Enum declaration in plain Verilog.
- Inconsistent FSM state transitions

Fixes were mostly similar and required detailed prompting. My testbenches confirmed the neuron spiked correctly and the SPI interface programmed expected values.

7. Can you think of any improvements to their solution?

- Replaced duplicated neuron code with generate blocks.
- Shared a single register file for neuron parameters.
- Improved testbench for input spike stream simulation.
- Used constants for FSM states instead of enums.
- Provided reusable module headers with parameterized widths.

8. Can you do a design with a RLU instead of a LIF neuron?

Yes, I created a ReLU-like neuron module:

```
module relu_neuron (
    input clk,
    input reset,
    input [7:0] input_current,
    output reg [7:0] output_current
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            output_current <= 8'd0;
        else
            output_current <= (input_current > 8'd0) ? input_current : 8'd0;
    end
```

endmodule

This model removed the temporal dynamics but made debugging easier. It's also closer to how modern artificial neurons work in ANNs.

9. Learnings throughout this challenge?

This assignment was an eye-opener for how AI can assist in hardware design. Here are my main learnings:

- LLMs are great for starting code but still need strong domain knowledge to refine and debug.
- Precision in prompting determines the quality of output significantly.
- Even a small Verilog error (like un-declared ports) can break synthesis, so careful review is critical.
- The design-debug loop is faster with AI, but verification and synthesis issues still require human judgment.
- Learned about neural models like LIF and ReLU and how they translate into logic-level hardware

Generated Verilog Code by ChatGPT during replication of the paper

LIF Neuron

```
module leaky_integrate_fire_neuron (
    input clk,
    input reset,
    input [7:0] current,
    input [7:0] THRESHOLD,
    input [7:0] LEAK_RATE,
    input [7:0] REFRAC_PERIOD,
    output reg spike
);
    reg [7:0] membrane_potential = 8'd0;
    reg [7:0] refrac_counter = 8'd0;
    reg in_refrac = 0;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            membrane_potential <= 8'd0;
            refrac_counter <= 8'd0;
            in_refrac <= 0;
            spike <= 0;
        end else begin
            spike <= 0;
            if (in_refrac) begin
                refrac_counter <= refrac_counter - 1;
```

```
            if (refrac_counter == 0) in_refrac <= 0;
        end else begin
            if (membrane_potential < LEAK_RATE)
                membrane_potential <= current;
            else if (membrane_potential + current < membrane_potential)
                membrane_potential <= THRESHOLD;
            else
                membrane_potential <= membrane_potential + current - LEAK_RATE;

            if (membrane_potential >= THRESHOLD) begin
                membrane_potential <= 0;
                spike <= 1;
                in_refrac <= 1;
                refrac_counter <= REFRAC_PERIOD;
            end
        end
    end
endmodule
```

SPI Module

```
module spi_slave (
    input clk,
    input reset,
    input sclk,
    input cs_n,
    input mosi,
    output reg miso,
    output reg [3:0] addr_out,
    output reg [7:0] data_out,
    output reg write_enable
);
    reg [1:0] state, next_state;
    reg [15:0] shift_reg;
    reg [7:0] addr_reg;
    integer bit_count;

    parameter IDLE = 2'b00, TRANSFER = 2'b01, COMPLETE = 2'b10;

    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= IDLE;
        else
            state <= next_state;
```

```
    end

    always @(posedge clk) begin
        case (state)
            IDLE: if (!cs_n) begin bit_count <= 15; next_state = TRANSFER; end
            TRANSFER: begin
                shift_reg[bit_count] <= mosi;
                if (bit_count == 0)
                    next_state = COMPLETE;
                else
                    bit_count <= bit_count - 1;
            end
            COMPLETE: if (cs_n) begin
                next_state = IDLE;
                addr_reg <= shift_reg[15:8];
                data_out <= shift_reg[7:0];
                write_enable <= 1;
            end
        endcase
    end

    always @(posedge clk) begin
        if (state == TRANSFER)
            miso <= shift_reg[15];
    end

    always @(*) begin
        addr_out = addr_reg;
        if (state != COMPLETE)
            write_enable = 0;
    end
endmodule
```

Network Module

```
module two_layer_network (
    input clk,
    input reset,
    input [7:0] input_spikes [2:0],
    output [7:0] output_spikes [2:0]
);
    wire [7:0] inter_spikes [2:0];
    wire [7:0] weights [2:0][2:0];
```

```
    genvar i;
    generate
        for (i = 0; i < 3; i = i + 1) begin : L1
            relu_neuron n (
                .clk(clk),
                .reset(reset),
                .input_current(input_spikes[i]),
                .output_current(inter_spikes[i])
            );
        end
    endgenerate

    genvar j;
    generate
        for (j = 0; j < 3; j = j + 1) begin : L2
            reg [7:0] sum;
            integer k;
            always @(*) begin
                sum = 0;
                for (k = 0; k < 3; k = k + 1)
                    sum = sum + (inter_spikes[k] * weights[k][j]);
            end

            relu_neuron out (
                .clk(clk),
                .reset(reset),
                .input_current(sum),
                .output_current(output_spikes[j])
            );
        end
    endgenerate
endmodule
```