4. Connections are made via **programmable synapses (weights)**

🧠 Design Overview

- Let's call it `spiking_nn_2layer`
- Parameters:
  - `N_INPUT`: number of input layer neurons
  - `N_OUTPUT`: number of output layer neurons
  - `WIDTH`: bit-width of all fixed-point signals
- Each connection has a programmable `weight`
- When an input neuron spikes, the weight is added to the target neuron in layer 2

✅ SystemVerilog Code

```systemverilog
module spiking_nn_2layer #(
    parameter int N_INPUT  = 4,
    parameter int N_OUTPUT = 3,
    parameter int WIDTH    = 8
)(
    input  logic                 clk,
    input  logic                 rst_n,
    input  logic [N_INPUT-1:0]   input_spikes,            // external spikes into layer 1
    output logic [N_OUTPUT-1:0]  output_spikes            // output spikes from layer 2
);

    // --- Synapse Weights (Programmable)
    logic [WIDTH-1:0] synapse_weights [N_INPUT-1:0][N_OUTPUT-1:0];

    // --- Output layer input accumulators
    logic [WIDTH-1:0] output_currents [N_OUTPUT-1:0];

    // --- Output Neurons
    lif_neuron #(
        .WIDTH(WIDTH),
        .THRESHOLD(100),
        .LEAK(2),
        .REF_PERIOD(5)
    ) output_layer [N_OUTPUT-1:0] (
        .clk(clk),
        .rst_n(rst_n),
        .input_current(output_currents),
        .spike(output_spikes)
    );

    // --- Synapse Computation Logic
    always_comb begin
        // Reset all output layer current inputs
        for (int j = 0; j < N_OUTPUT; j++) begin
            output_currents[j] = 0;
        end

        // For each input neuron that spiked, add its weight to each output neuron
        for (int i = 0; i < N_INPUT; i++) begin
            if (input_spikes[i]) begin
                for (int j = 0; j < N_OUTPUT; j++) begin
                    output_currents[j] += synapse_weights[i][j];
                end
```

```
            end
        end
    end

endmodule
`
```

🧠 Notes:

- `input_spikes` could come from actual `lif_neuron` modules or be injected from a testbench/stimulus.
- Synapse weights (`synapse_weights`) are fully programmable (via `initial` block or external interface).
- Each spike from layer 1 adds its corresponding weight to every neuron in layer 2 → classic **fully connected** behavior.

Would you like me to:

- Add input layer neurons as actual `lif_neuron` modules?
- Make synapse weights externally writeable (e.g., via an AXI-lite or memory-mapped interface)?
- Create a testbench to demonstrate it?

explain how this code works

Absolutely! Let's walk through how the `spiking_nn_2layer` SystemVerilog module works — section by section — so you get a clear picture of the architecture and functionality.

🧠 Overall Purpose

This module creates a **2-layer spiking neural network**, where:

- Layer 1 receives **spike inputs** (`input_spikes`)
- Layer 2 consists of `lif_neuron` modules (your previously defined spiking neuron)
- Every input neuron is **fully connected** to every output neuron through a programmable **synapse weight**
- A spike in input neuron `i` adds `synapse_weights[i][j]` to output neuron `j`'s current

---

🔧 Module Parameters

```
systemverilog

`parameter int N_INPUT  = 4,
parameter int N_OUTPUT = 3,
```

```
parameter int WIDTH    = 8
```

These parameters control:

- `N_INPUT` = number of input neurons (first layer)
- `N_OUTPUT` = number of output neurons (second layer)
- `WIDTH` = bit width of current signals and weights (e.g., 8 bits)

🔌 Inputs and Outputs

```systemverilog
input  logic [N_INPUT-1:0]   input_spikes,    // Incoming spikes (e.g., from other neurons or stim
output logic [N_OUTPUT-1:0]  output_spikes    // Output spikes from layer 2 neurons
```

- `input_spikes`: 1-bit signal per neuron in layer 1. High (1) = neuron fired a spike.
- `output_spikes`: Each output neuron spikes (1) when its membrane potential reaches threshold.

🔗 Synapse Weights

```systemverilog
logic [WIDTH-1:0] synapse_weights [N_INPUT-1:0][N_OUTPUT-1:0];
```

- A 2D array of weights connecting **every input neuron to every output neuron**.
- This models a **fully connected network**.
- You can initialize or update these weights from a higher-level controller or `initial` block.

🔋 Output Currents

```systemverilog
logic [WIDTH-1:0] output_currents [N_OUTPUT-1:0];
```

- Temporary storage for the total current input to each neuron in layer 2.
- Reset to zero each cycle, then populated based on incoming spikes and synapse weights.

🧠 Layer 2 Neurons

```systemverilog
`lif_neuron output_layer [N_OUTPUT-1:0] (...);
`
```

- This creates an **array of LIF neurons**, one for each output.
- Each receives its own `` `output_currents[j]` ``, and produces `` `output_spikes[j]` ``.

⚙️ Spike-to-Current Conversion Logic

```systemverilog
`always_comb begin
    // Reset output currents
    for (int j = 0; j < N_OUTPUT; j++) begin
        output_currents[j] = 0;
    end

    // Loop through all spiking input neurons
    for (int i = 0; i < N_INPUT; i++) begin
        if (input_spikes[i]) begin
            for (int j = 0; j < N_OUTPUT; j++) begin
                output_currents[j] += synapse_weights[i][j];
            end
        end
    end
end
`
```

This is where the **spike propagation happens**:

- If `` `input_spikes[i]` `` is high (neuron `` `i` `` fired), then:
    - Its weight to **each** output neuron `` `j` `` is added to that neuron's `` `output_currents[j]` ``
- In short: every spike "fans out" to the next layer, scaled by a weight

---

🔁 Per Cycle Behavior

On every clock cycle:

1. `` `input_spikes` `` are evaluated
2. For each spiking input neuron, its weights are added to the corresponding output neuron's input current
3. Each output neuron updates its `` `membrane_potential` `` and may spike if threshold is reached

---

✅ Summary

This module gives you:

- A clean 2-layer spiking neural net

- Fully programmable weights (can be used for learning or static connectivity)
- Modular structure using your existing LIF neuron design

Want to:

- Simulate it with test data?
- Expand to 3+ layers?
- Add dynamic weight updates or STDP?

Just say the word!

so it basically has one layer of neurons or 2? give me some clarity

Great question — and totally fair. The confusion comes from how we define "layers" in a spiking neural network (SNN), especially when the input layer might not contain actual neurons.

Let's clear this up:

🔍 In the Code You Have:

Layer 1: Input layer

- This layer **does not instantiate LIF neurons**.
- It expects **spike signals** (`` `input_spikes` ``) to be driven from *outside* the module — e.g.,:
  - Another neural layer
  - Sensor input
  - Testbench stimulus

So it's more like a **stimulus source** than a modeled neuron layer.

Layer 2: Output layer

- This is the only actual instantiated layer of **LIF neurons**.
- It consists of `` `N_OUTPUT` `` neurons.
- Each neuron receives current from all `` `N_INPUT` `` sources, scaled by programmable `` `synapse_weights` ``.

🔁 Connections:

- Every input spike goes through a weight → summed into an output neuron's input current
- This forms a **fully connected projection** from input to output

---

🧠 So How Many "Neuron Layers"?

✅ Current design has:

- **1 real neuron layer** (Layer 2 — the output LIF neurons)
- **1 abstract input layer**, made up of spike sources

📌 If You Want 2 Actual Neuron Layers:

Then you need to:

1. Instantiate **LIF neurons** in the **input layer** too
2. Drive them with external input currents
3. Use **their spikes** to feed the second layer

I can help you write that version too — want to move in that direction?

That makes sense

Perfect — glad that clarified it! You've got a solid grasp on this now.

Let me know if/when you want to:

- Add **LIF neurons to the input layer** (making it a full 2-layer neuron network),
- Simulate some spike trains and watch the network fire,
- Make weights **trainable or programmable at runtime**, or
- Expand toward a larger architecture (e.g., convolutional spiking network or temporal coding).

I'm here for all of it 🧠⚡

so we are adding the weightahe instead of multiplying it to get the inopuut current for layer 2?

🔥 Yes — and you've spotted another key design point in spiking neural networks.

Let's unpack it clearly:

🧠 Are We Adding or Multiplying the Weights?

✅ We are multiplying, but in a very sparse, spike-driven way.

In the code:

```systemverilog
`if (input_spikes[i]) begin
    output_currents[j] += synapse_weights[i][j];
end
`
```

This is **functionally equivalent to**:

```text
`output_current[j] += input_spikes[i] * synapse_weights[i][j]
`
```

Because:

- `` `input_spikes[i]` `` is a **binary spike**: 0 (no spike) or 1 (spike)
- Multiplying a value by 0 or 1 is the same as **conditioned addition**

🔬 Why This Makes Sense in SNNs

In a spiking neural network:

- Spikes represent **discrete events in time**, not continuous activations
- You don't feed forward a floating-point value — you **emit a spike**, and when that spike occurs, it **delivers a weight** to downstream neurons

So the logic:

```systemverilog
`if (spike) send weight
`
```

is how **weighted spike transmission** works.

🧮 Mathematically Speaking

For output neuron `` `j` ``, input current at time `` `t` `` is:

$$I_j(t) = \sum_{i=0}^{N_{in}-1} w_{ij} \cdot s_i(t)$$

Where:

- w_{ij}: synapse weight from input neuron `` `i` `` to output neuron `` `j` ``
- $s_i(t) \in \{0, 1\}$: spike from neuron `` `i` `` at time `` `t` ``

That's exactly what your code is doing — just optimized into an `` `if` `` instead of a `` `*` ``.

✅ **TL;DR:**

- Yes, we **multiply weights by spike values**, but since spikes are binary, it's implemented as **conditioned addition**.
- It's standard and totally correct for SNNs.

Let me know if you want to experiment with:

- Analog-valued spike encodings (e.g., rate coding)
- Signed weights
- Synaptic delays or decay dynamics

why temporary storage for current?

Awesome question again — you're really thinking like a hardware designer now. 🧠⚡

Let's get into **why we use temporary storage (`output_currents`)** for the second layer's input current before passing it to the neurons.

🗃️ 1. Temporal Computation

```
systemverilog

`always_comb begin
    for (int j = 0; j < N_OUTPUT; j++)
        output_currents[j] = 0;

    for (int i = 0; i < N_INPUT; i++) begin
        if (input_spikes[i]) begin
            for (int j = 0; j < N_OUTPUT; j++) begin
                output_currents[j] += synapse_weights[i][j];
            end
        end
    end
end
`
```

This block:

- **Collects contributions** from all input spikes
- Sums them up for each output neuron
- This all happens in a single cycle — a **combinational calculation**

So we need a place to store the **accumulated values** before passing them into the LIF neurons — hence, `output_currents`.

📦 2. Fan-in Accumulation

Each output neuron receives input from **multiple sources**. For example:

If neuron `0` in the second layer is connected to `4` input neurons, it needs to **sum** up 4 weights (if all 4 spike).

Without a temporary variable:

- You'd either overwrite data
- Or need a much more complex handshaking/memory interface

🧠 3. Cleaner Separation of Computation and Neuron Dynamics

- `output_currents[j]` is the **input** to the neuron
- The neuron uses that current and decides whether to spike or not
- Keeping current calculation **outside** the neuron helps you:
  - Decouple spike delivery from membrane dynamics
  - Modularize and scale cleanly

💡 Analogy

Think of it like:

> A neuron is waiting at the mailbox. All incoming spikes drop their weighted "mail" into the box. When the clock ticks, the neuron checks the box (current input), updates its internal state, and maybe spikes.

✅ TL;DR

We use temporary `output_currents` because:

- We need to **accumulate** multiple weighted inputs per output neuron
- It cleanly separates **spike routing logic** from **neuron dynamics**
- It makes the design more modular, readable, and scalable

Want to take this next into pipelining, weight update logic, or a multi-layer version?

now give me the lif neuron and are these weights being passed as parameter or as input?

Great combo of questions — let's tackle both parts cleanly:

🧠 Part 1: Are the weights passed as parameters or inputs?