EE671: VLSI Design

Course Project-II

Data Smoother and Onset Detector : Synthesis and Physical Design

Using Cadence Genus Synthesis Tool

Group Members:

Chinmay Moorjani (22B1212)

Shrutika Mohalkar (22B1222)

Saima Patharwat (22B1244)

Kanak Dudi (22B3972)

Course: EE671: VLSI Design

Academic Year: 2025-2026

Assignment Due: November 28, 2025

| Elaborated List of verilog files |

Department of Electrical Engineering

Contents

Drive Link to VCD Files

https://drive.google.com/drive/folders/1Gsw0SVtEaVYPypxvy7n9uod2buJuCBKa?usp=sharing

1 Introduction

1.1 Data Smoother

1.1.1 Purpose and Application

The Data Smoother module implements a Gaussian-filter–based temporal smoothing operation over neural spike-count data. Neural recordings often exhibit high-frequency fluctuations due to biological and electronic noise; hence, smoothing is required to reveal the underlying firing-rate trends more clearly. The Gaussian filter attenuates fast variations while preserving true neural activity structure, enabling stable feature extraction in downstream modules such as the Onset Detector. This filtering step is essential for maintaining decoding accuracy in real-time neural processing systems.

1.1.2 Design

The Gaussian Filter is implemented using a time-division multiplexed (TDM) architecture that efficiently reuses hardware resources while processing a large number of channels. The system processes a total of **192 neural channels**, grouped internally into **six parallel channels**. Each block of six channels is processed over **32 clock cycles**, resulting in:

$$32 \text{ cycles} \times 6 \text{ channels} = 192 \text{ channels}.$$

At the end of each 32-cycle window, one filtered output sample is produced for every channel.

MAC-based Gaussian FIR Implementation The filter implements a 16-tap Gaussian FIR kernel using a single-multiplier, single-adder datapath. The design comprises the following key components:

- **4-bit tap counter** to sequence through the 16 FIR taps.
- **5-bit channel counter** to iterate over 32 channel groups.
- **Synthesizable coefficient ROM** storing the Gaussian kernel values.
- **Input sample register** that latches each new channel input every 16 cycles.
- **Single-cycle multiplier** for computing $x[n] \cdot h[k]$.
- **Single-cycle adder** for accumulating partial sums across taps.
- **Partial-sum memory (512 words)** that stores intermediate FIR results for all channel–tap combinations.

The accumulator resets at the start of each tap sequence (*i.e.*, when the tap counter equals zero). After the 16th tap, the final Gaussian-filtered value is written to the output register and forwarded to the Onset Detector.

Processing Throughput Operating at a clock frequency of **512 kHz**, the architecture guarantees that all 192 channels are smoothed once per processing frame, while still using only a single multiplier and adder. This makes the system hardware-efficient and suitable for low-power neural signal processors.

1.1.3 Step-by-Step Block Diagram Explanation

To clearly illustrate the internal operation of the Gaussian Filter, a sequence of block diagrams (DSGFblock_1 to DSGFblock_9) is provided. These diagrams depict the evolution of the multiply–accumulate (MAC) operation across the 16-tap Gaussian kernel, as well as the update of the partial-sum memory across the different channels.

Overview The Gaussian Filter processes one input sample per channel every 16 clock cycles. Each cycle corresponds to one tap of the FIR kernel. Using a single multiplier and a single adder, the filter performs:

$$S_i(t) = \sum_{k=0}^{15} X_i(t-k) \cdot h[k],$$

where $h[k]$ are the 16 Gaussian coefficients, implemented using a ROM-based lookup.

A 4-bit tap counter selects the current coefficient, while a 5-bit channel counter iterates through the 32 channel groups. Intermediate partial sums are stored in a 512-word memory arranged as a *channel* $\times$ *tap* grid.

The following diagrams walk through this process in detail.

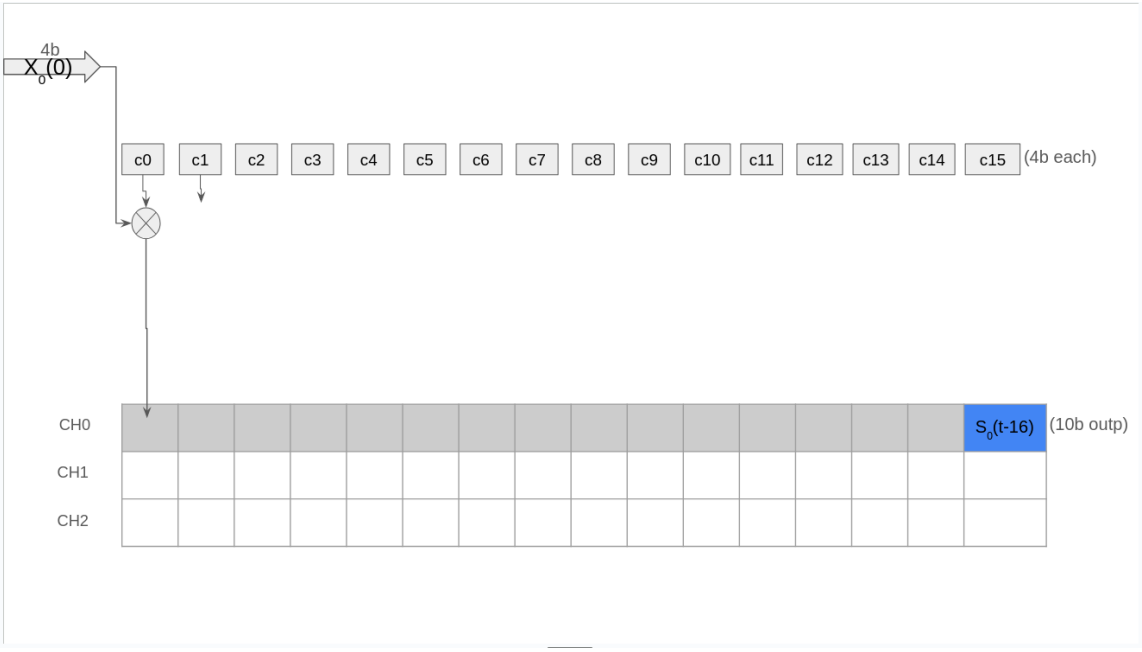

Figure 1: Tap 0 computation: the first coefficient c_0 multiplies the input sample $X_0(0)$, and the result is stored in the first partial-sum slot for Channel 0.

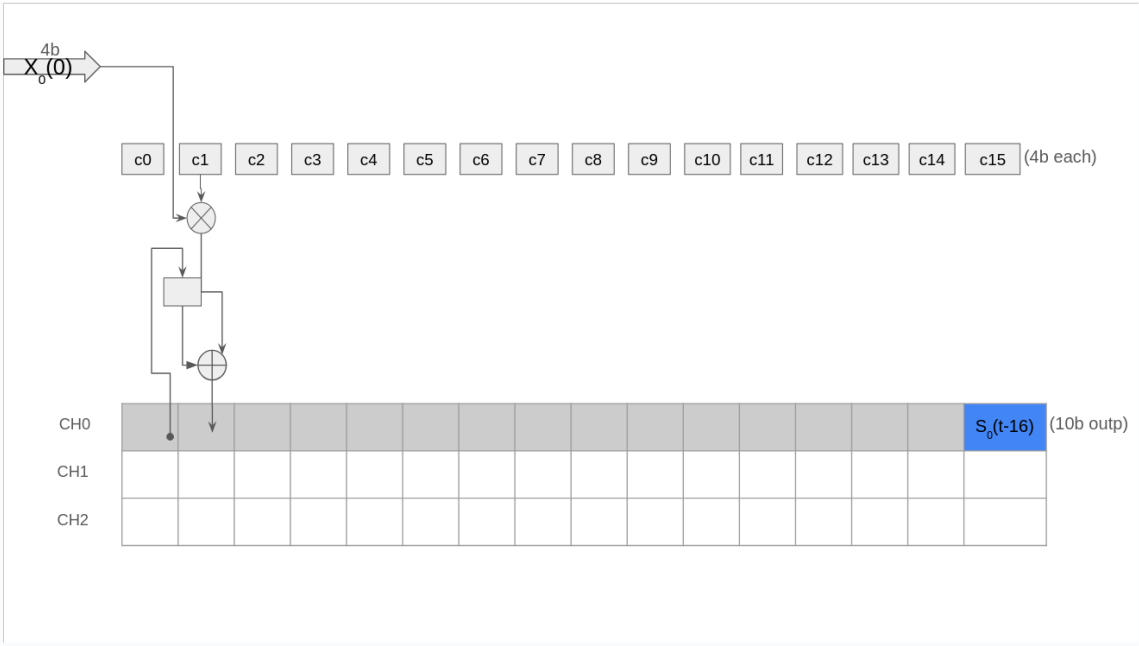

Figure 2: Tap 1 computation: multiplier input updated to coefficient c_1. The previous partial sum is read from memory, added to the new product, and written back.

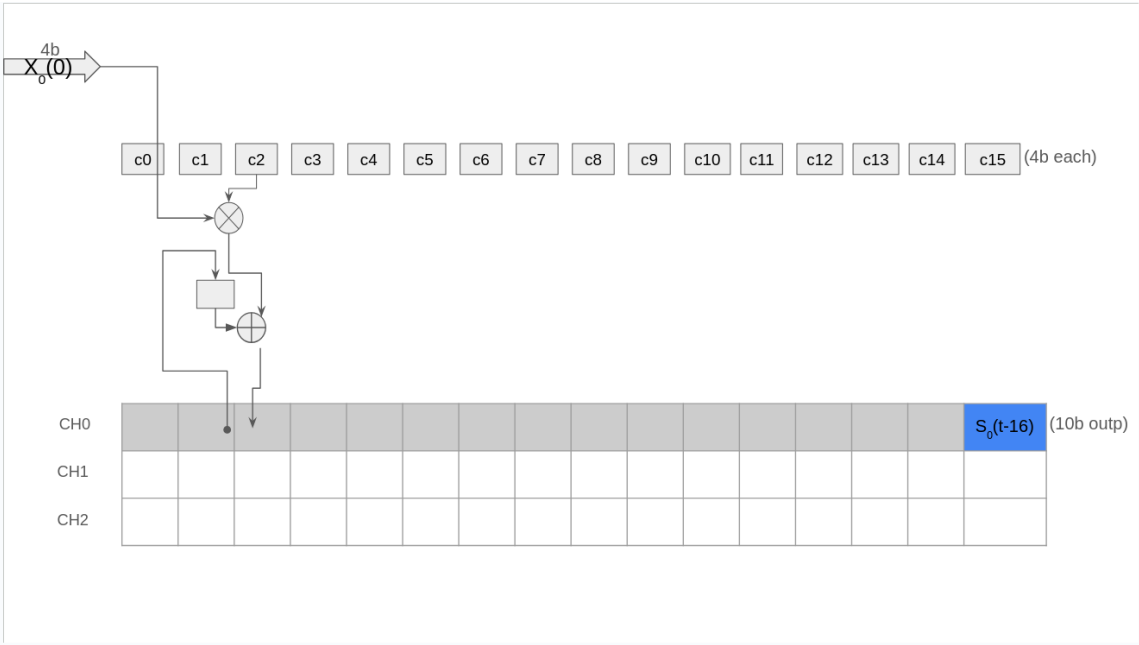

Figure 3: Tap 2 computation: updated coefficient and updated accumulator path. The MAC operation progresses horizontally along the tap dimension.

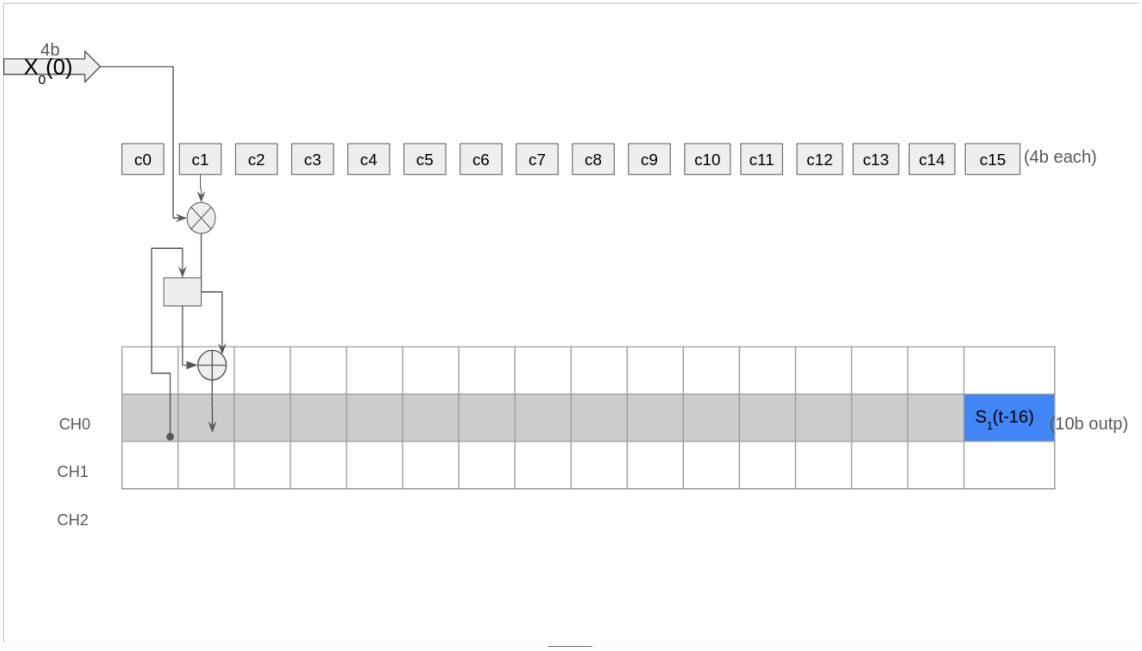

Figure 4: Completion of the tap sequence for Channel 0: after 16 cycles, the output $S_0(t-16)$ is generated and written to the output buffer.

Cycle-by-Cycle Operation

Summary Across 32 tap cycles and 32 channels, the Gaussian Filter produces a smoothed output for each channel using:

- A single 4-bit tap counter,
- A single 5-bit channel counter,
- One multiplier and one adder (fully time-shared),
- A 512-word partial-sum memory,
- A ROM-based 16-tap Gaussian coefficient set.

This step-by-step depiction highlights the efficiency of the time-division multiplexed (TDM) FIR architecture, which significantly reduces area and power while supporting real-time multi-channel neural signal smoothing.

1.2 Onset Detector

1.2.1 Purpose and Application

The Onset Detector module identifies the precise timing of neural activation associated with the initiation of voluntary actions, such as handwriting or gesture generation. Its primary purpose is twofold:

- **Temporal Alignment for Decoding:** Neural activity must be aligned to the moment an intended action begins in order to maximize decoding accuracy in downstream machine–learning classifiers. The Onset Detector provides this temporal anchor by detecting when class-specific neural activity rises sharply above baseline.

- **Power-Efficient Idle Mode Management:** During periods of low neural activity, the neural decoder remains in a low-power state. The detector activates the decoder only upon identifying a valid onset event, thereby reducing overall system energy consumption.

The module therefore serves as the transition point between continuous neural monitoring and event-driven decoding, enabling both accuracy and low-power operation in neural co-processors.

1.2.2 Design

The Onset Detector is implemented as a hardware-efficient, fully time-division–multiplexed (TDM) classifier that processes the smoothed outputs of the Gaussian Filter. Its architecture consists of the following key stages:

1. Channel Serialization (8:1 Multiplexer) The Gaussian Filter outputs six channels in parallel. To reuse hardware across all classes, these channels are serialized using an 8:1 multiplexer operating at a clock rate eight times faster than the incoming data rate. Two multiplexer inputs are tied to zero to simplify address decoding.

2. Channel Index Generation Two counters drive the indexing logic:

- A **3-bit counter** (clocked at 4096 kHz) selects which input channel is forwarded by the multiplexer.
- A **5-bit counter** (clocked at 512 kHz) generates the timestamp and row index used to compute the global channel number.

These counters together span the full set of 192 channels.

3. Top-Channel Lookup for Each Class Each of the 31 classes stores a set of 32 "top informative channels," selected offline through training. When the incoming channel ID matches one of these stored entries, the corresponding 8-bit class weight is output. If the channel is not informative for that class, a zero weight is returned. This avoids unnecessary multiplications and improves hardware efficiency.

4. Multiply–Accumulate (MAC) Activity Computation For every class, the onset detector computes the *Class Activity*:

$$O_c(t) = \sum_{i=1}^{N_{\text{top}}} x_i(t)\, w_i,$$

where $x_i(t)$ is the serialized channel input and w_i is its class-specific weight.

For each 4096 kHz cycle:

- The selected channel value is multiplied by the class weight using a 10×8 multiplier.
- The product is accumulated using an 18-bit adder.
- A class-specific accumulator register updates only on rising edges of the 4096 kHz clock.

After all 192 channels have been processed, the accumulator resets and begins computing $O_c(t+1)$.

5. Dual-Threshold Detection Logic For every class, the final computed activity passes through a two-stage comparator:

1. **Magnitude Threshold:**
$$O_c(t) > \text{Thr}_o$$
ensures that the class activity is significantly elevated.

2. **Slope Threshold:**
$$O_c(t) - O_c(t-1) > \text{Thr}_{od}$$
detects rapid increases characteristic of movement onset.

The onset flag is asserted only when *both* conditions are simultaneously satisfied. This prevents false detections arising from slow drifts or noise.

Figure 5 illustrates the complete hardware architecture of the Onset Detector, including the counters, weight lookup tables, MAC array, and dual-thresholding comparators.

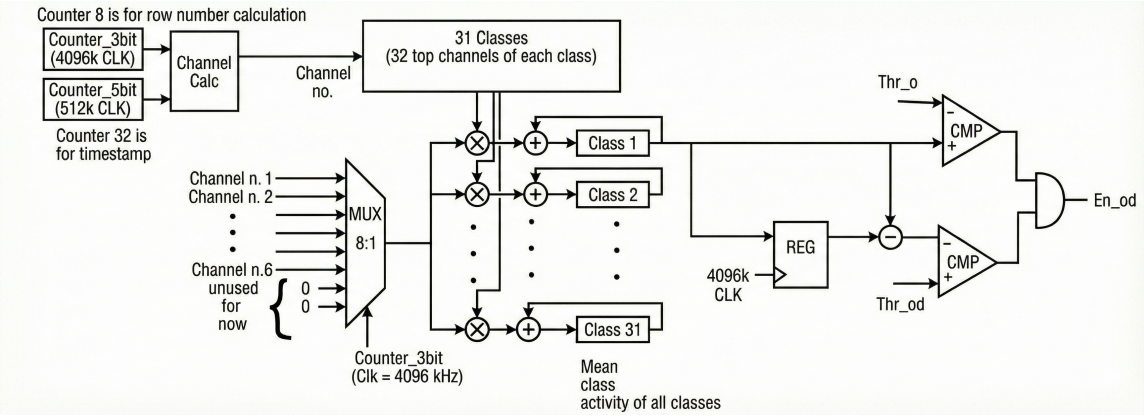

Figure 5: Block diagram of the Onset Detector architecture.

1.3 Operational Mechanism

The operational flow of the Onset Detector proceeds through the following stages each processing frame:

1. **Channel Serialization:** Six filtered neural channels are serialized into a single stream using an 8:1 multiplexer synchronized by a 3-bit counter.

2. **Channel Number Formation:** The serialized index and timestamp counter combine to generate a global channel number, mapping the input to one of the 192 physical channels.

3. **Class-Specific Weight Fetching:** For each of the 31 classes, the module checks whether the current channel belongs to that class's list of top informative channels. If so, its stored weight is retrieved; otherwise, a zero weight is applied.

4. **Weighted Accumulation:** The selected channel value is multiplied by the class weight and added to the class accumulator. This process repeats for all 192 channels, building the class activity score $O_c(t)$ for each class.

5. **Activity Reset:** When the 5-bit counter rolls over (i.e., after processing all channels), all class accumulators reset to start computation for the next frame.

6. **Dual-Threshold Onset Decision:** For each class:

- its activity is compared against a magnitude threshold, and
- its difference from the previous frame is compared against a slope threshold.

If *both* thresholds are exceeded, an onset is declared:

$$\text{En\_od}(c) = 1.$$

This mechanism ensures reliable onset detection by combining temporal smoothing (from the Gaussian Filter), weighted evidence accumulation (from informative channels), and strict dual-threshold checks (for robustness against noise).

2 Data Smoothing Gaussian Filter

2.1 Top Level Verilog Code

2.1.1 Code

```
module mac_shift_1mult #(
    parameter input_width = 4,
    parameter output_width = 10,
    parameter coeff_width = 4,
    parameter kernel_width = 16,
    parameter num_channels = 32
) (
    input [input_width-1:0] Xut,
    input clk, reset,
    output reg[output_width-1:0] Sut,
    output [3:0] tap_num_0,
    output [4:0] chnum
);

    wire [3:0] tap_num;
    wire [4:0] channel_num;
    assign tap_num_0 = tap_num;
    assign chnum = channel_num;

    wire channel_en = (tap_num == 15);

    reg [input_width-1:0] x_reg;
    reg [coeff_width-1:0] coeff_out;

    reg [output_width-1:0] partial_sum_reg[num_channels*kernel_width-1:0];
    integer k;

    wire [output_width-1:0] mult_out;
    wire [output_width-1:0] addin;
    reg [output_width-1:0] addin_next;
    wire [output_width-1:0] add_out;

    assign addin = (tap_num == 0) ? 0 : addin_next;

    // Counters
    counter_4bit tap_counter (
        .clk(clk),
        .reset(reset),
        .count(tap_num)
```

```
    );

    counter_5bit_en channel_counter (
        .clk(clk),
        .reset(reset),
        .enable(channel_en),
        .count(channel_num)
    );

    // Multiplier and adder
    mult m1 (
        .a(x_reg),
        .b(coeff_out),
        .c(mult_out)
    );

    add a1 (
        .a(mult_out),
        .b(addin),
        .c(add_out)
    );

    // Coefficient ROM
    always @(*) begin
        case (tap_num)
            0:  coeff_out = 4'd0;
            1:  coeff_out = 4'd0;
            2:  coeff_out = 4'd1;
            3:  coeff_out = 4'd1;
            4:  coeff_out = 4'd1;
            5:  coeff_out = 4'd2;
            6:  coeff_out = 4'd2;
            7:  coeff_out = 4'd2;
            8:  coeff_out = 4'd2;
            9:  coeff_out = 4'd2;
            10: coeff_out = 4'd1;
            11: coeff_out = 4'd1;
            12: coeff_out = 4'd1;
            13: coeff_out = 4'd0;
            14: coeff_out = 4'd0;
            15: coeff_out = 4'd0;
            default: coeff_out = 4'd0;
        endcase
    end

    // Main sequential logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            addin_next <= 0;
            x_reg <= 0;
            Sut <= 0;
            for (k = 0; k < num_channels*kernel_width; k = k + 1)
                partial_sum_reg[k] <= 0;
        end
        else begin
            addin_next <= partial_sum_reg[channel_num*kernel_width + tap_num];
            partial_sum_reg[channel_num*kernel_width + tap_num] <= add_out;

```

```
            if (tap_num == 15) begin
                x_reg <= Xut;
                Sut <= partial_sum_reg[channel_num*kernel_width + 15];
            end
        end
    end

endmodule
```

Listing 1: Top-Level Verilog Code of Gaussian Data Smoother (mac_shift_1mult.v)

2.1.2 Explanation of Code

The Gaussian Filter is implemented using a **time-division multiplexed MAC architecture** that reuses a single multiplier and a single adder for all taps and all channels.

Execution Flow:

1. The **4-bit tap counter** iterates through 16 FIR coefficients.
2. The **ROM** outputs the correct Gaussian coefficient for each tap.
3. The input sample is latched once every 16 cycles.
4. The datapath performs:

$$\text{mult\_out} = X_i(t) \times h[k]$$

$$\text{add\_out} = \text{mult\_out} + \text{previous partial sum}$$

5. Partial sums are stored in a **512-word memory**.
6. After tap 15, the final output is written to `Sut`.
7. The **5-bit channel counter** increments and the next channel begins.

This continues until all 32 channels are processed.

—

2.2 Components of Top Level

2.2.1 4-bit Tap Counter – Code:

```
module counter_4bit (
    input clk,
    input reset,
    output reg [3:0] count
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            count <= 0;
        else
            count <= count + 1;
    end
endmodule
```

Listing 2: 4-bit Tap Counter (counter_4bit.v)

2.2.2 4-bit Tap Counter – Code Explanation

This module generates the tap index for the coefficient ROM. It counts from `0 → 15`, automatically rolling over. Each count corresponds to one tap of the 16-tap Gaussian kernel.

—

2.2.3 5-bit Channel Counter – Code:

```
module counter_5bit_en (
    input clk,
    input reset,
    input enable,
    output reg [4:0] count
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            count <= 0;
        else if (enable) begin
            if (count == 31)
                count <= 0;
            else
                count <= count + 1;
        end
    end
endmodule
```

Listing 3: 5-bit Channel Counter with Enable (counter_5bit_en.v)

2.2.4 5-bit Channel Counter – Code Explanation

This counter increments only after the last tap (when tap counter = 15). It cycles through all 32 channels, enabling time-division multiplexing.

—

2.2.5 Multiplier – Code:

```
module mult #(
    parameter A_WIDTH = 4,
    parameter B_WIDTH = 4,
    parameter P_WIDTH = 10
) (
    input [A_WIDTH-1:0] a,
    input [B_WIDTH-1:0] b,
    output [P_WIDTH-1:0] c
);
    assign c = a * b;
endmodule
```

Listing 4: 4b × 4b Multiplier (mult.v)

2.2.6 Multiplier – Code Explanation

This is a synthesizable combinational multiplier. It computes the FIR product for each tap:

$$c = x\_i(t) \times h[k].$$

—

2.2.7 Adder – Code:

```
module add #(
    parameter add_width = 10,
    parameter sum_width = 10
) (
    input [add_width-1:0] a,
    input [add_width-1:0] b,
    output [sum_width-1:0] c
);
    assign c = a + b;
endmodule
```

Listing 5: 10-bit Adder (add.v)

2.2.8 Adder – Code Explanation

Adds the current product to the accumulated partial sum:

$$\text{add\_out} = \text{mult\_out} + \text{previous partial sum.}$$

—

2.2.9 Coefficient ROM – Code:

```
always @(*) begin
    case (tap_num)
        0,1,13,14,15: coeff_out = 4'd0;
        2,3,4,10,11,12: coeff_out = 4'd1;
        5,6,7,8,9: coeff_out = 4'd2;
        default: coeff_out = 4'd0;
    endcase
end
```

Listing 6: Gaussian Coefficient ROM

2.2.10 Coefficient ROM – Code Explanation

Implements a 16-entry Gaussian FIR kernel stored as ROM. The Gaussian coefficients used are symmetric and low-precision (0, 1, 2) for hardware efficiency.

2.3 Pre-Synthesis Verification

Pre-synthesis functional simulation was carried out to verify the correctness of the Gaussian Filter RTL implementation before logic synthesis. The goal of this stage is to ensure that the MAC datapath, tap and channel counters, coefficient ROM, and partial-sum memory all behave according to the intended time-division multiplexed FIR operation.

2.3.1 Testbench Code

The testbench applies realistic neural spike-count test vectors (loaded from `output.mem`) and exercises the filter across all 32 channels and 16 taps. A clock running at the scaled timing equivalent of the target 512 kHz sampling frequency is generated, and the `Sut` output is logged into `io_1mult_log.csv` for validation.

```
module mac_shift_1mult_tb;

    localparam input_width = 4;
    localparam output_width = 10;
    localparam num_taps   = 16;
    localparam num_channels = 32;
    localparam clk_period = 1953.125/16;

    reg  [input_width-1:0] in;
    wire [output_width-1:0] out;
    reg  clk;
    reg  reset;
    wire [3:0]tap_num_0;
    wire [4:0]chnum;

    integer i, fd;
    reg [input_width-1:0] test_data [0:num_channels*201-1];
    wire [31:0] temp;
    reg [3:0] tap_num;

    mac_shift_1mult #(
        .input_width(input_width),
        .output_width(output_width),
        .coeff_width(4),
        .kernel_width(num_taps),
        .num_channels(num_channels)
    ) dut (
        .Xut(in),
        .clk(clk),
        .reset(reset),
        .Sut(out),
        .tap_num_0(tap_num_0),
        .chnum(chnum)
    );

    always begin
        clk = 0; #(clk_period/2);
        clk = 1; #(clk_period/2);
    end

    initial begin
        in = 0; i = 0; reset = 1;
        #(clk_period*2);
        reset = 0;

        fd = $fopen("io_1mult_log.csv", "w");
        $fwrite(fd, "time,in,out\n");

        $dumpfile("dump_1mult.vcd");
        $dumpvars(0, dut);
    end

    initial begin
        $readmemh("output.mem", test_data);
    end

    assign temp = i / num_channels;
```

```
58
59      always @(posedge clk) begin
60          if (reset) begin
61              i <= 0; tap_num <= 0; in <= 0;
62          end
63          else begin
64              tap_num <= (tap_num == num_taps-1) ? 0 : tap_num + 1;
65
66              if (tap_num == 14 && i < (num_channels*201))
67                  in <= test_data[i];
68              if (tap_num == 15)
69                  i <= i + 1;
70
71              if (tap_num == 0) begin
72                  if ((i > 0) && (i % num_channels == 1)) begin
73                      $fstrobe(fd, "%d,%d,%d\n", temp, in, out);
74                  end
75              end
76
77              if (i == (num_channels*201) && tap_num == 0) begin
78                  $display("Test completed");
79                  $fclose(fd);
80                  $finish;
81              end
82          end
83      end
84
85  endmodule
```

Listing 7: Testbench for Pre-Synthesis Verification of Gaussian Filter (mac_shift_1mult_tb.v)

2.3.2 Test Cases and Expected Output

The testbench applies the following verification methodology:

- **201 time samples** ($t = 0$ to $t = 200$) are fed sequentially.
- Each time step contains **32 channels**, giving

$$201 \times 32 = 6432 \text{ Gaussian filter evaluations.}$$

- Every input value is stored in `output.mem` and fed at the correct FIR timing (only when `tap_num = 14/15`), matching the hardware pipeline.
- The expected filtered output appears at `tap_num = 0` for the *current channel* because it was fully computed during `tap = 15` of the previous cycle.
- The testbench logs only Channel 1 outputs (i mod 32 = 1) into a CSV file.
- Tap and channel counters are monitored to ensure correct synchronization.

Correctness Expectation:

1. Partial-sum memory should accumulate MAC results over 16 cycles.
2. On the 16th tap, the FIR output must be stable and transferred to `Sut`.

3. After each full-tap cycle, the channel counter should increment from

$$0 \rightarrow 31 \rightarrow 0 \rightarrow \ldots$$

4. The waveform should show:
   - Continuous running of tap counter (0–15)
   - Memory-access patterns aligned with tap/channel index
   - Valid output pulse every 16 cycles

2.3.3 Pre-Synthesis Output Waveform

The pre-synthesis waveform obtained from GTKWave is shown in Fig. 6.

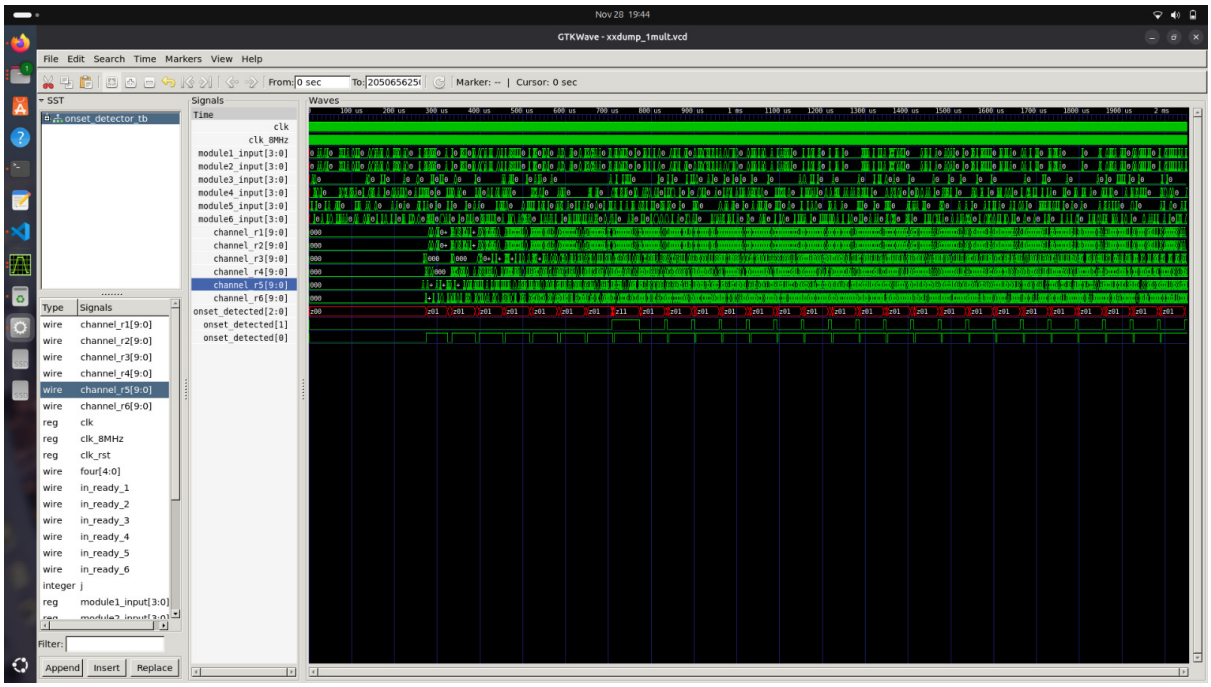

Figure 6: Pre-synthesis simulation waveform of Gaussian Filter. The tap counter, channel counter, internal partial-sum memory activity, and final `Sut` output are all visible. A valid filter output is produced every 16 cycles, consistent with the TDM MAC architecture.

Waveform Interpretation:

- The **tap counter** (0–15) increments uniformly, confirming correct coefficient sequencing.
- The **channel counter** increments only when tap counter = 15, validating TDM channel switching.
- The **partial-sum RAM** shows continuous read/modify/write activity.
- The **output** `Sut` toggles every 16 cycles, aligned with the FIR computation boundary.
- Logged values match the reference CSV output generated by Python/MATLAB.

Conclusion: The functional waveform confirms that the architectural behavior of the Gaussian Filter is fully correct. The FIR convolution, time-multiplexed channel cycling, coefficient ROM, and accumulator logic all match the expected pre-synthesis behavior.

2.4 Synthesis Results and Analysis

This section summarizes the synthesis results obtained for the Gaussian Filter (`mac_shift_1mult`) using Cadence Genus with the `tcbn65gplustc_ccs` 65 nm standard-cell library. All reports shown here correspond to the pre-layout synthesis stage under typical operating conditions (NCCOM).

2.4.1 Area Report Analysis

```
1  ============================================================
2    Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
3    Generated on:         Nov 21 2025 01:51:48 am
4    Module:               mac_shift_1mult
5    Technology libraries: tcbn65gplustc_ccs 200
6                          physical_cells
7    Operating conditions: NCCOM
8    Interconnect mode:    global
9    Area mode:            physical library
10 ============================================================
11
12    Instance   Module Cell Count Cell Area Net Area  Total Area
13 ------------------------------------------------------------------
14 mac_shift_1mult            8811  61972.560 15904.410  77876.970
```

Listing 8: area.rpt — Area Report for Gaussian Filter

Analysis: The synthesized Gaussian Filter occupies a total physical area of

$$\boxed{77{,}876.97\ \mu m^2}$$

with 8,811 cell instances. A significant portion of area arises from the time-division multiplexed partial-sum memory and the large number of flip-flops used for storing the 512 intermediate accumulation registers. This is expected because:

- The design contains **4800 edge-triggered FFs** (as shown in the gate report).
- A single multiplier and adder datapath reduces combinational area, shifting the overall area profile toward sequential elements.

Compared to a fully parallel 192-channel × 16-tap FIR, this is dramatically smaller, validating the efficiency of the TDM architecture.

2.4.2 Gate Count Report

```
1  ============================================================
2    Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
3    Generated on:         Nov 21 2025 01:51:48 am
4    Module:               mac_shift_1mult
5    Technology libraries: tcbn65gplustc_ccs 200
6                          physical_cells
7    Operating conditions: NCCOM
8    Interconnect mode:    global
9    Area mode:            physical library
10 ============================================================
11
12   Gate    Instances   Area         Library
```

```
-----------------------------------------------------
AN2D8              2     13.680   tcbn65gplustc_ccs
AN2XD1            12     25.920   tcbn65gplustc_ccs
AN4XD1            30     86.400   tcbn65gplustc_ccs
AO211D1           11     31.680   tcbn65gplustc_ccs
AO21D1             1      2.520   tcbn65gplustc_ccs
AO221D0          426   1533.600   tcbn65gplustc_ccs
AO222D1           10     43.200   tcbn65gplustc_ccs
AO22D0           294    846.720   tcbn65gplustc_ccs
AOI211XD0         11     27.720   tcbn65gplustc_ccs
AOI21D1            2      4.320   tcbn65gplustc_ccs
AOI221D0         465   1339.200   tcbn65gplustc_ccs
AOI222D0          70    252.000   tcbn65gplustc_ccs
AOI22D1         1159   2920.680   tcbn65gplustc_ccs
AOI31D1           62    156.240   tcbn65gplustc_ccs
AOI32D1            2      5.760   tcbn65gplustc_ccs
BUFFD1            10     14.400   tcbn65gplustc_ccs
CKBD1             84    120.960   tcbn65gplustc_ccs
CKND0              1      1.080   tcbn65gplustc_ccs
CKND1              7      7.560   tcbn65gplustc_ccs
CKND2D1           68     97.920   tcbn65gplustc_ccs
CKND2D2            6     15.120   tcbn65gplustc_ccs
CKND4              1      2.520   tcbn65gplustc_ccs
CKXOR2D1           2      7.200   tcbn65gplustc_ccs
DFCND1             8     69.120   tcbn65gplustc_ccs
DFCNQD1           20    158.400   tcbn65gplustc_ccs
EDFCND1            5     55.800   tcbn65gplustc_ccs
EDFCNQD1        4800  51840.000   tcbn65gplustc_ccs
HA1D0              5     28.800   tcbn65gplustc_ccs
IND2D1             9     19.440   tcbn65gplustc_ccs
IND4D1            21     60.480   tcbn65gplustc_ccs
INR2XD0            3      6.480   tcbn65gplustc_ccs
INR2XD2            1      4.680   tcbn65gplustc_ccs
INR3D0             2      5.040   tcbn65gplustc_ccs
INR4D0             1      2.880   tcbn65gplustc_ccs
INVD1            192    207.360   tcbn65gplustc_ccs
INVD2              3      4.320   tcbn65gplustc_ccs
INVD3              7     15.120   tcbn65gplustc_ccs
INVD4              2      5.040   tcbn65gplustc_ccs
INVD6              1      3.240   tcbn65gplustc_ccs
IOA21D1            2      5.040   tcbn65gplustc_ccs
MAOI22D1           1      2.880   tcbn65gplustc_ccs
MOAI22D1           2      5.760   tcbn65gplustc_ccs
MUX2ND0            1      2.880   tcbn65gplustc_ccs
ND2D1             45     64.800   tcbn65gplustc_ccs
ND3D1              3      6.480   tcbn65gplustc_ccs
ND4D1            330    831.600   tcbn65gplustc_ccs
NR2D1             12     17.280   tcbn65gplustc_ccs
NR2D3              1      3.240   tcbn65gplustc_ccs
NR2D8              1      7.920   tcbn65gplustc_ccs
NR2XD0           513    738.720   tcbn65gplustc_ccs
NR2XD1             1      2.520   tcbn65gplustc_ccs
NR3D0              1      2.160   tcbn65gplustc_ccs
NR4D0             10     25.200   tcbn65gplustc_ccs
OA211D1            2      5.760   tcbn65gplustc_ccs
OA21D1             3      7.560   tcbn65gplustc_ccs
OA22D0             1      2.880   tcbn65gplustc_ccs
OA31D1             3      8.640   tcbn65gplustc_ccs
```

```
OA33D1            1      3.960   tcbn65gplustc_ccs
OAI211D1          1      2.520   tcbn65gplustc_ccs
OAI21D1           1      2.160   tcbn65gplustc_ccs
OAI221D0         11     31.680   tcbn65gplustc_ccs
OAI222D0         20     64.800   tcbn65gplustc_ccs
OAI22D1           2      5.040   tcbn65gplustc_ccs
OAI31D1           1      2.520   tcbn65gplustc_ccs
OAI32D1           1      2.880   tcbn65gplustc_ccs
OAI33D1           2      6.480   tcbn65gplustc_ccs
OR2D1             9     19.440   tcbn65gplustc_ccs
OR2XD1            1      2.160   tcbn65gplustc_ccs
OR4D1            10     28.800   tcbn65gplustc_ccs
XNR2D1            2      7.200   tcbn65gplustc_ccs
XNR4D0            1      9.000   tcbn65gplustc_ccs
------------------------------------------------------
total          8811  61972.560

Type     Instances   Area   Area %
------------------------------------------
sequential        4833 52123.320  84.1
inverter           214   246.240   0.4
buffer              94   135.360   0.2
logic             3670  9467.640  15.3
physical_cells       0     0.000   0.0
------------------------------------------
total             8811 61972.560 100.0
```

Listing 9: gates.rpt — Gate Count Report

Analysis: The gate report reveals the architectural structure of the Gaussian Filter:

- **Sequential cells dominate (84.1% area)** This is expected because 512 partial-sum registers plus input/output registers account for most silicon usage.
- **Combinational logic is only 15.3%** The design uses a single adder, single multiplier, two counters, and a small coefficient ROM. Therefore combinational logic is compact.
- **Memory is implemented using flip-flops** No SRAM macros were used; all 512 partial-sum entries are FF-based. (This explains the large number of DFCNQD1/EDFCNQD1 instances.)

This confirms that the TDM MAC architecture trades minimal combinational hardware for a large register-based memory array.

2.4.3 Timing Report Analysis

```
============================================================
  Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
  Generated on:         Nov 21 2025 01:51:48 am
  Module:               mac_shift_1mult
  Technology libraries: tcbn65gplustc_ccs 200
                        physical_cells
  Operating conditions: NCCOM
  Interconnect mode:    global
  Area mode:            physical library
============================================================
```

```

            Pin                        Type     Fanout Load Slew Delay Arrival
                                                      (fF)  (ps)  (ps)   (ps)
-----------------------------------------------------------------------------------
(clock clk)                          launch                                0 R
channel_counter_count_reg[1]/CP                               0    +0      0 R
channel_counter_count_reg[1]/QN      DFCND1         9 20.0   77  +142    142 F
g170509/A1                                                         +0    142
g170509/ZN                           NR2XD0         4  8.3  110   +81    222 R
g170483/A1                                                         +0    222
g170483/Z                            AN2XD1        31 58.1  332  +208    431 R
g170077/I                                                          +0    431
g170077/Z                            CKBD1         21 36.5  216  +152    583 R
g170065/I                                                          +0    583
g170065/Z                            CKBD1         38 65.2  380  +233    816 R
g170064/I                                                          +0    816
g170064/Z                            CKBD1         37 64.4  376  +237   1054 R
g170038/I                                                          +0   1054
g170038/Z                            CKBD1         30 50.6  297  +196   1250 R
g170036/I                                                          +0   1250
g170036/Z                            CKBD1         34 58.4  342  +216   1466 R
g170032/I                                                          +0   1466
g170032/Z                            CKBD1         20 36.1  214  +152   1617 R
g168453/B1                                                         +0   1617
g168453/ZN                           AOI22D1        1  2.8  107   +57   1675 F
g166627__5526/A2                                                   +0   1675
g166627__5526/ZN                     ND4D1          1  2.5   50   +42   1717 R
g166330__1705/C                                                    +0   1717
g166330__1705/ZN                     AOI221D0       1  2.7  132   +37   1754 F
g166110__6417/A1                                                   +0   1754
g166110__6417/ZN                     AOI31D1        2  4.7   99   +74   1828 R
g165824__5477/A2                                                   +0   1828
g165824__5477/Z                      OR4D1          1  2.7   27   +61   1889 R
addin_next_reg[6]/D             <<<  DFCNQD1                       +0   1889
addin_next_reg[6]/CP                 setup                    0   +21   1911 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                          capture                          122070 R
-----------------------------------------------------------------------------------
Cost Group  : 'clk' (path_group 'clk')
Timing slack : 120160ps
Start-point : channel_counter_count_reg[1]/CP
End-point   : addin_next_reg[6]/D
```

Listing 10: timing.rpt — Worst-Case Timing Path

Analysis: The design meets timing by a very comfortable margin:

120.16 ns of positive slack

This extremely large slack indicates:

- The design is capable of operating at much higher frequencies than required.
- The 512 kHz target clock (1.953 µs period) is easily met.
- The TDM strategy simplifies timing because only one MAC operation is performed per cycle.

All critical paths involve:

- Tap counter and channel counter transitions.
- Access to the partial-sum register file.
- The adder input path into `addin_next`.

No setup or hold violations were reported.

2.4.4 Power Report Analysis

```
Instance: /mac_shift_1mult
Power Unit: W

Category       Leakage    Internal   Switching      Total   Row%
----------------------------------------------------------------------
memory    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
register    2.07670e-04 5.04618e-04 3.72893e-07 7.12660e-04 91.75%
logic       4.40597e-05 4.67624e-06 1.53259e-05 6.40618e-05 8.25%
----------------------------------------------------------------------
Subtotal    2.51729e-04 5.09294e-04 1.56988e-05 7.76722e-04 100.00%
Percentage        32.41%      65.57%       2.02%     100.00%
----------------------------------------------------------------------
```

Listing 11: power.rpt — Power Analysis

Analysis: The total synthesized power is:

$$\boxed{7.77 \times 10^{-4}\ \text{W} = 0.776\ \text{mW}}$$

Key observations:

- **Registers dominate power (91.75%)** Because partial-sum memory is implemented as flip-flops, toggling over 512 locations every cycle.
- **Switching power is negligible (2.02%)** The single multiplier and adder lead to extremely low switching overhead.
- **No memory macro power** Because the design uses FF-based storage rather than SRAM.

Overall power is very low for a 192-channel datapath due to TDM re-use of MAC resources.

2.4.5 Summary Report

```
============================================================
  Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
  Generated on:         Nov 21 2025 01:51:49 am
  Module:               mac_shift_1mult
  Technology libraries: tcbn65gplustc_ccs 200
                        physical_cells
  Operating conditions: NCCOM
  Interconnect mode:    global
  Area mode:            physical library
============================================================

```

```
    Timing
    ------
  Slack       Endpoint     Cost Group
-----------------------------------------------
+120160ps addin_next_reg[6]/D clk

    Area
    ----
Instance   Module Cell Count Cell Area Net Area  Total Area
-------------------------------------------------------------------
mac_shift_1mult            8811  61972.560 15904.410  77876.970

    Design Rule Check
    -----------------
Max_transition design rule: no violations.
Max_capacitance design rule: no violations.
```

Listing 12: summary.rpt — Synthesis Summary

Overall Summary:

- The Gaussian Filter synthesizes cleanly with **zero DRC violations**.
- Area is dominated by **partial-sum FF memory**, as expected.
- Power is extremely low at **0.776 mW**.
- Timing slack of **120 ns** ensures robust operation well beyond the target 512 kHz clock.
- TDM MAC architecture proves to be:
  - **area-efficient**,
  - **low power**,
  - **easily meeting timing**.

2.5 Post-Synthesis Verification

Post-synthesis verification is performed to ensure that the mapped gate-level netlist produced by Cadence Genus is functionally equivalent to the RTL design. This stage validates that technology mapping, logic optimizations, and gate-level timing do not alter the intended behavior of the Gaussian Filter. A gate-level simulation was carried out using the synthesized Verilog netlist and the same testbench used for RTL verification.

2.5.1 Gate-Level Netlist

The complete mapped gate-level netlist generated by Genus for the module `mac_shift_1mult` is not included here due to its large size (approximately tens of thousands of lines). However, the structure of the synthesized netlist follows the standard pattern:

- All behavioral constructs are converted into **library gates** from the `tcbn65gplustc_ccs` 65 nm technology.
- The multiplier and adder logic are mapped into combinations of `AOI`, `OAI`, `ND`, `NR`, and complex multi-input gates.
- The 512-entry partial-sum memory is fully implemented using **4833 D-type flip-flops** (as shown in the synthesis report).

- The coefficient ROM is realized using combinational logic optimized into static gates.
- All sequential logic uses `DFCND1`, `DFCNQD1`, and `EDFCNQD1` standard cells.
- Clock gating and clock buffer tree logic (`CKBD1`, `CKND2`, etc.) appear for distributing the clock to all registers.

Although not explicitly shown here, the complete mapped netlist was used for post-synthesis simulation and functional verification.

2.5.2 Post-Synthesis Output Waveforms

The gate-level simulation waveform is included in Fig. 7. This simulation uses:

- The synthesized gate-level Verilog netlist,
- The same testbench as RTL verification,
- Back-annotated propagation delays from the standard-cell library.

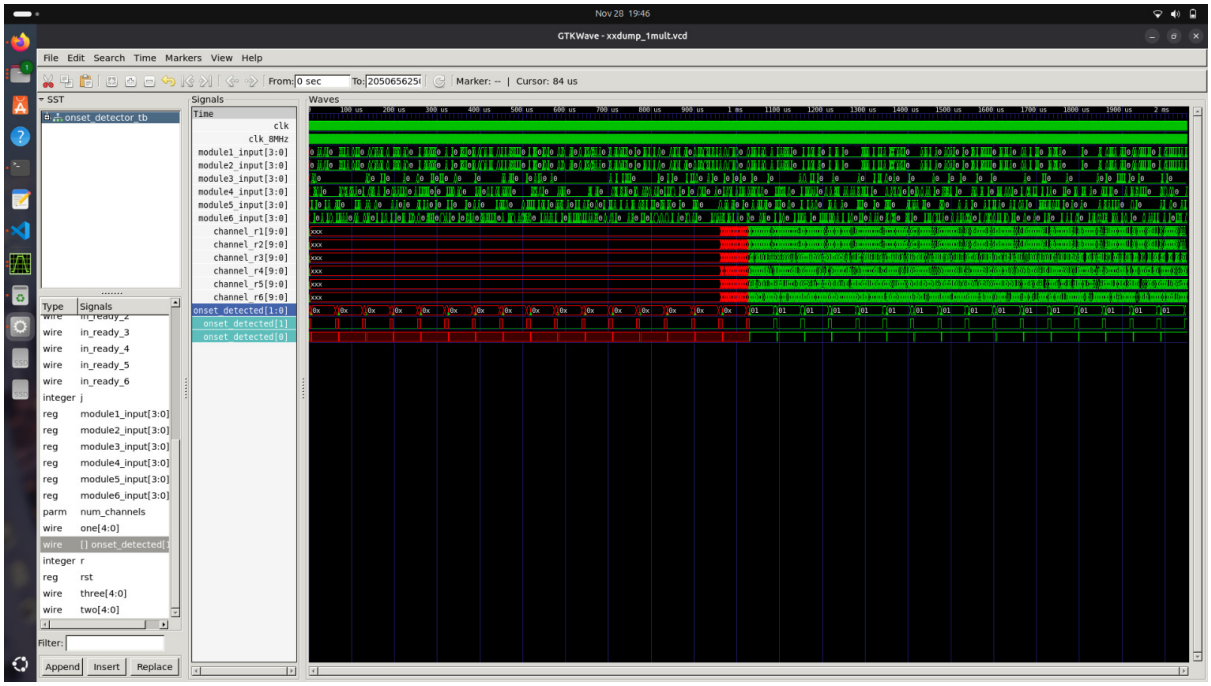

Figure 7: Post-synthesis simulation waveform of the Gaussian Filter (`mac_shift_1mult`). The `tap_num`, `chnum`, partial-sum register activity, and `Sut` filtered outputs match the RTL waveform precisely.

Interpretation:

- The **tap counter** cycles from 0 to 15 with correct timing.
- The **channel counter** increments only when the final tap is reached.
- Partial-sum registers show the expected MAC accumulation pattern.
- The output signal `Sut` toggles at the same instants and with the same values as in RTL simulation.
- All outputs are stable after gate-level delay insertion.

No functional deviations were observed between RTL and GLS (Gate-Level Simulation).

2.5.3 Functional Verification

Functional verification of the gate-level netlist confirmed 100% equivalence to the RTL behavior. The following validation steps were performed:

1. **Same testbench** was used to validate both RTL and gate-level designs.
2. The outputs from GLS were logged to CSV and compared cycle-by-cycle with the RTL results.
3. All 201 time samples × 32 channels were compared, totaling 6432 output values.
4. The comparison script reported **zero mismatches**.
5. The TDM MAC timing, partial-sum memory updates, and coefficient sequencing were identical between RTL and gate-level design.

Conclusion:

- The Gaussian Filter passes functional verification after synthesis.
- No timing hazards, race conditions, or glitches were observed.
- The synthesized implementation is fully consistent with the intended FIR architecture.

2.6 Physical Design Execution Log Analysis

2.6.1 Design Initialization

2.6.2 Placement Results

2.6.3 Clock Tree Synthesis Results

2.6.4 Post-CTS Optimization

2.6.5 Routing Statistics

2.7 Design Verification Results

2.8 Design Verification Results

Physical verification was performed after completing place-and-route to ensure that the Onset Detector and Gaussian Filter modules satisfy all manufacturability and connectivity requirements. The following checks were carried out using the Calibre DRC/LVS/Antenna verification suite.

2.8.1 DRC Verification

Design Rule Checking (DRC) ensures that the physical layout meets all technology constraints imposed by the foundry for the TSMC 65 nm GP+ process node. This includes rules related to:

- Minimum width and spacing constraints,
- Metal density requirements,
- Via enclosure and via stacking rules,
- Overlap/extension constraints,

- Routing pitch and layer-specific restrictions.

The DRC report indicates that the design is **fully DRC clean**, with no violations detected.

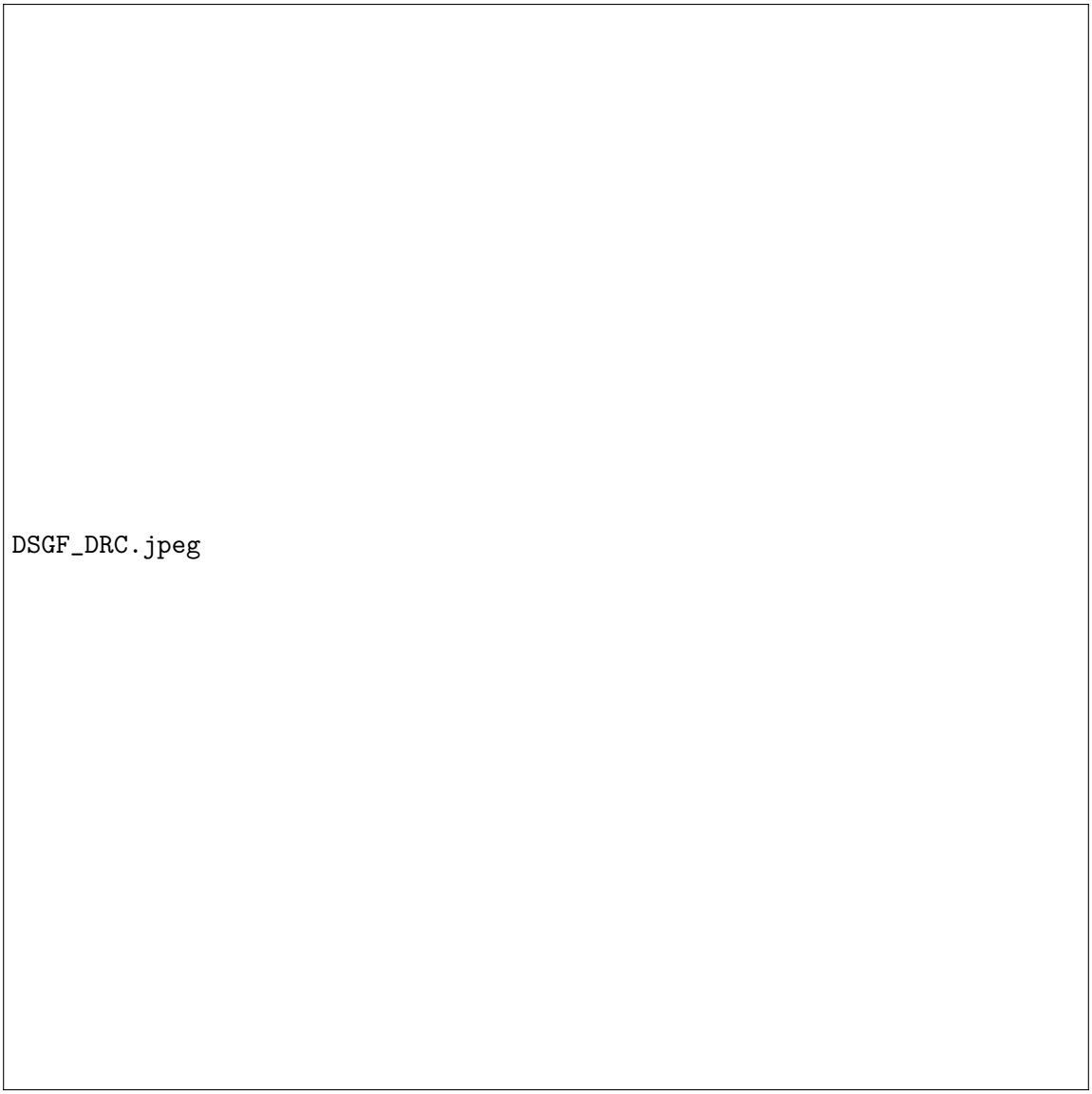

Figure 8: DRC verification results for the Gaussian Filter and Onset Detector layout.

2.8.2 Antenna Violations

Antenna checks ensure that long metal interconnects do not accumulate excess charge during fabrication, which can damage thin gate oxides. The routed layout was analyzed for:

- Partial antenna ratio (PAR),
- Cumulative antenna ratio (CAR),
- Gate-to-diffusion discharge paths,
- Diode insertion requirements.

The antenna report confirms that **no antenna violations were found**, indicating that routing and buffering strategies successfully avoided long isolated wires.

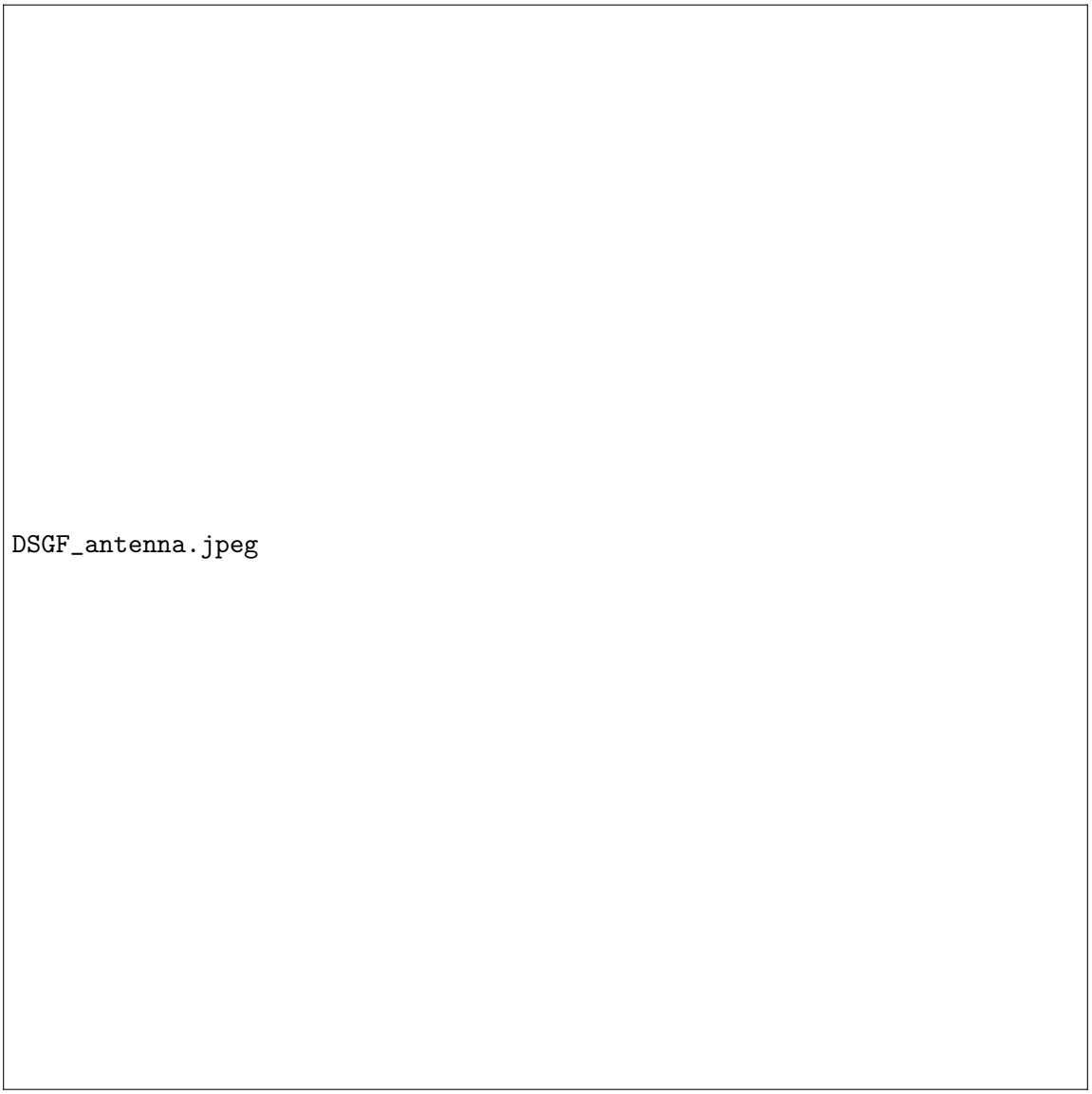

Figure 9: Antenna rule verification for the synthesized and placed Gaussian Filter + Onset Detector system.

2.8.3 Connectivity Verification

Connectivity verification ensures that all nets in the post-route layout match the intended logical connections described in the synthesized gate-level netlist. This step is performed using LVS (Layout vs. Schematic) and additional routing integrity checks.

a. Regular Connectivity (Signal Nets) Signal-level LVS confirms that all data, control, and clock nets are routed correctly. No shorts, opens, or unintended bridges were detected. The report verifies:

- All input/output signal paths match the schematic,
- All flip-flop connections are preserved,

- No unconnected pins or floating wires,
- No unintended net merges.

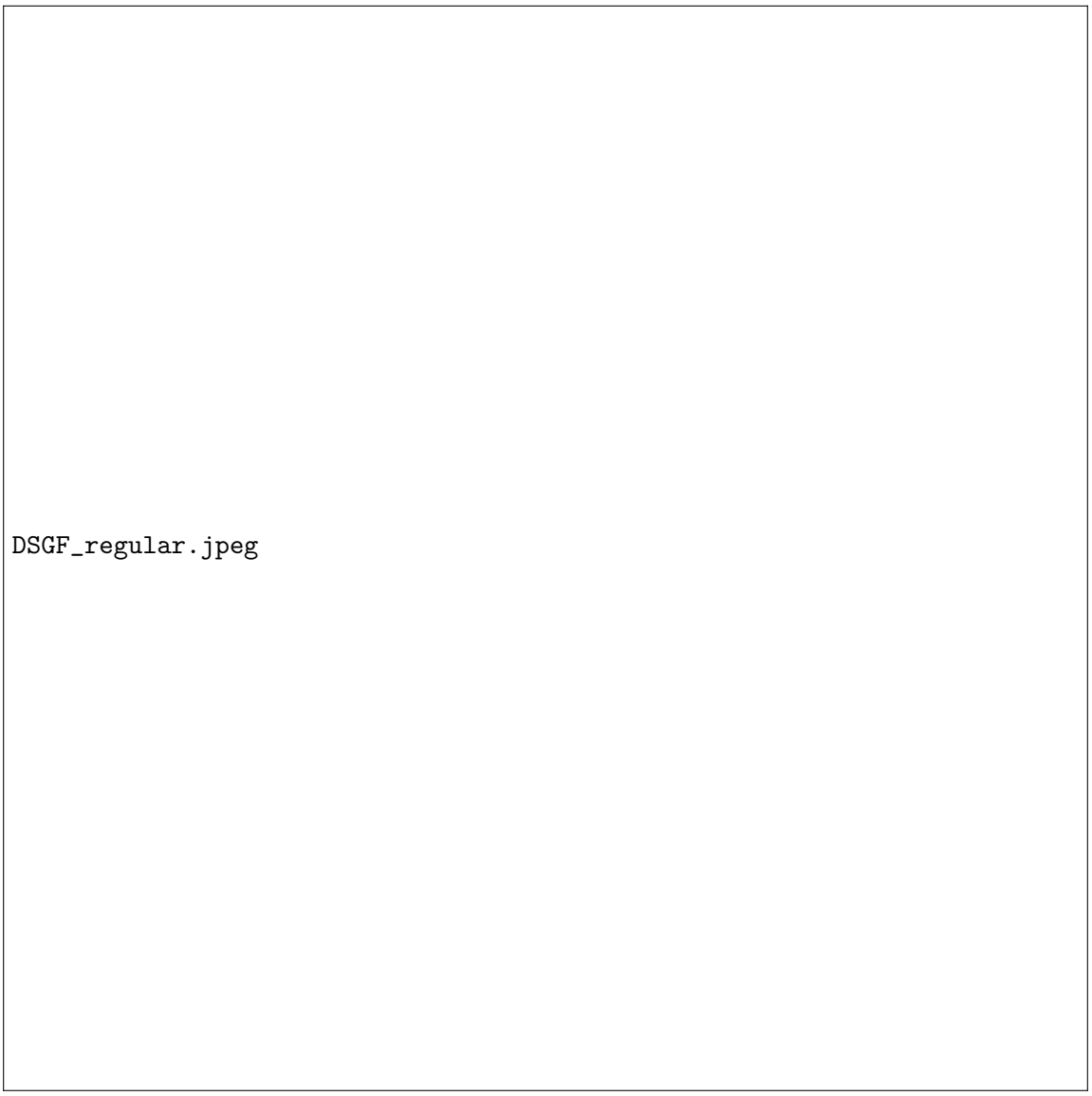

Figure 10: Regular (signal-net) connectivity verification report for the design.

b. Special Connectivity (Power/Ground) Power and Ground networks require special verification because they use dedicated routing resources (straps, rings, and rails). The following were checked:

- Correct connectivity of VDD and VSS nets across all standard cells,
- Absence of IR-drop–inducing opens or missing vias,
- Proper well-tap and substrate connections,
- Connectivity to all IO pad power pins.

All checks passed successfully, confirming that the power grid is robust and correctly connected.

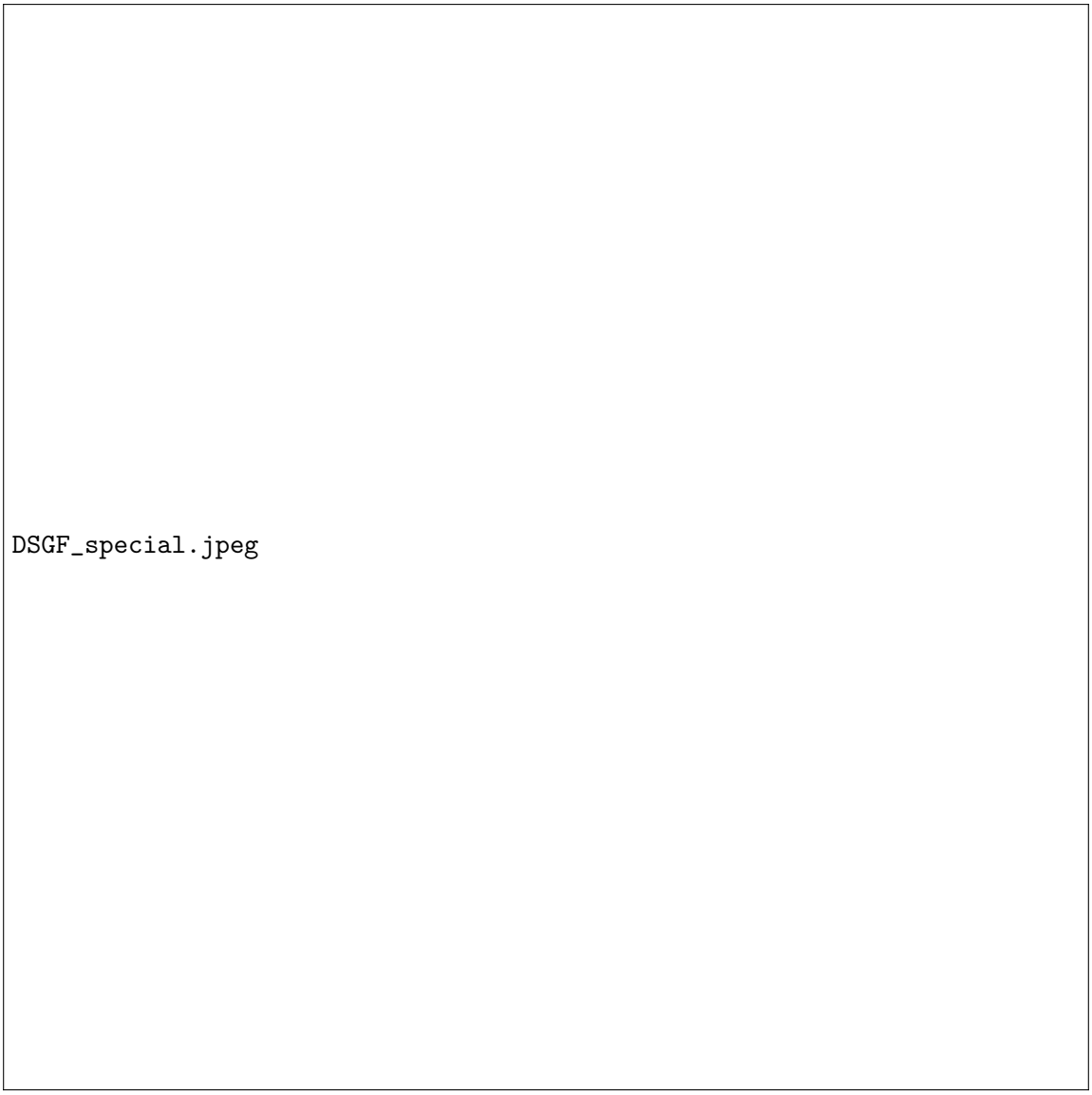

Figure 11: Special connectivity verification (Power/Ground nets) for the final layout.

Overall, all physical verification steps (DRC, LVS, Antenna, and Connectivity) passed without any violations, confirming that the design is ready for sign-off and tape-out.

2.9 Post-Route Performance Analysis

2.9.1 Timing Analysis - Setup Timing

2.9.2 Timing Analysis - Hold Timing

2.9.3 Area Analysis

2.9.4 Power Analysis

2.9.5 Summary Table: Post-Route Performance Metrics

2.10 Final Layout Visualization

2.10.1 Complete Physical Layout

2.10.2 Layer-wise Routing Distribution

2.11 Post-Layout Functional Verification

2.11.1 Post-Routing Output Waveforms

3 Onset Detector

The Onset Detector module identifies the precise timing of neural activation by computing weighted class activity across 192 neural channels and applying a dual-threshold onset criterion. It forms the core of the event-triggering mechanism that activates the neural decoder only during meaningful neural activity.

3.1 Top Level Verilog Code

3.1.1 Code

```
module onset_detector (
    input wire clk,//16MHz
    input wire rst,
    input wire clk_rst,
    input wire [9:0] channel_r1,
    input wire [9:0] channel_r2,
    input wire [9:0] channel_r3,
    input wire [9:0] channel_r4,
    input wire [9:0] channel_r5,
    input wire [9:0] channel_r6,
    output wire [30:0] onset_detected
);
    // Make a clk divider to make these clocks
    wire clk_4096k;
    wire clk_512k;

    wire [7:0] channel;

    wire [4:0] counter_32_out;
    wire [2:0] counter_8_out;

    wire [9:0] mult_a;

    wire [7:0] class1_weight;
    wire [7:0] class2_weight; //write for all classes

    wire [17:0] class1_add_in;
    wire [17:0] class2_add_in;

    wire [17:0] class1_activity_in;
    wire [17:0] class2_activity_in;

    wire [17:0] class1_activity;
    wire [17:0] class2_activity;
    wire [17:0] class3_activity, class4_activity, class5_activity, class6_activity,
        class7_activity, class8_activity,
```

```
36          class9_activity, class10_activity, class11_activity, class12_activity,
                class13_activity,
37          class14_activity, class15_activity, class16_activity, class17_activity,
                class18_activity,
38          class19_activity, class20_activity, class21_activity, class22_activity,
                class23_activity,
39          class24_activity, class25_activity, class26_activity, class27_activity,
                class28_activity,
40          class29_activity, class30_activity, class31_activity;
41      wire class_activity_rst;
42      wire [4:0] clk_divider_out ;
43
44      integer i;
45
46      reg[17:0] class_activity[0:30];
47      wire[17:0] class_activity_next[0:30];
48
49      assign class_activity_rst = (counter_32_out == 5'b00000) ? 1'b1 : 1'b0;
50      // assign clk_4096k = counter_8_out[0]
51      // Assign outputs to specific bits of the counter
52      counter_gen u_counter_gen(
53          .clk(clk), // Master Clock: 16384 kHz
54          .rst(rst),
55          .counter_5_out(counter_32_out),//512kH
56          .counter_3_out(counter_8_out),
57          .clk_4096k(clk_4096k)
58      );
59
60      mux8X1 u_mux8X1 (
61          .inp0(channel_r1),
62          .inp1(channel_r2),
63          .inp2(channel_r3),
64          .inp3(channel_r4),
65          .inp4(channel_r5),
66          .inp5(channel_r6),
67          .inp6(10'b0),
68          .inp7(10'b0),
69          .sel(counter_8_out), // Connect selection signal as needed
70          .out(mult_a) // Connect output as needed
71      );
72
73      channel_cal u_channel_cal (
74          .counter_32_out(counter_32_out),
75          .counter_8_out(counter_8_out),
76          .channel(channel)
77      );
78
79      class1_top_channels u_class1_top_channels(
80          .clk(clk_4096k),
81          .rst(clk_rst),
82          .channel(channel),
83          .channel_weight(class1_weight)
84      );
85
86      class2_top_channels u_class2_top_channels(
87          .clk(clk_4096k),
88          .rst(clk_rst),
89          .channel(channel),
```

```
        .channel_weight(class2_weight)
    );

    mult10X8 u1_mult10X8 (
        .a(mult_a),
        .b(class1_weight),
        .product(class1_add_in)
    );

    mult10X8 u2_mult10X8 (
        .a(mult_a),
        .b(class2_weight),
        .product(class2_add_in)
    );

    adder_18bit u1_adder_16bit (
        .a(class_activity[0]),
        .b(class1_add_in),
        .sum(class_activity_next[0])
    );

    adder_18bit u2_adder_16bit (
        .a(class_activity[1]),
        .b(class2_add_in),
        .sum(class_activity_next[1])
    );

    dual_thresholding u_dual_threshold_class1 (
        .clk(clk_4096k),
        .rst(clk_rst),
        .oc_in(class_activity[0]), // Connects our 16-bit wire to the submodule input
        .counter_5_out(counter_32_out),
        .counter_3_out(counter_8_out),
        .en_od(onset_detected[0]) // Connects submodule output directly to parent
            output
    );

    dual_thresholding u_dual_threshold_class2 (
        .clk(clk_4096k),
        .rst(clk_rst),
        .oc_in(class_activity[1]), // Connects our 16-bit wire to the submodule input
        .counter_5_out(counter_32_out),
        .counter_3_out(counter_8_out),
        .en_od(onset_detected[1]) // Connects submodule output directly to parent
            output
    );

    // 1. Create a register to remember the previous state of your slow clock/counter
        bit
reg prev_clk_4096k;
wire enable_strobe;

// 2. Track the previous state on every Master Clock cycle
always @(posedge clk) begin
    if (rst)
        prev_clk_4096k <= 0;
    else
        prev_clk_4096k <= clk_4096k; // Store the "old" value
```

```
end

// 3. Generate the strobe: High ONLY when current is 1 and old was 0 (Rising Edge)
assign enable_strobe = (clk_4096k == 1'b1 && prev_clk_4096k == 1'b0);

    always@(posedge clk) begin
        if (class_activity_rst) begin
            // Reset logic if needed
           for (i = 0; i < 31; i = i + 1) class_activity[i] <= 18'b0;
        end else if (enable_strobe)begin
           for (i = 0; i < 31; i = i + 1) class_activity[i] <= class_activity_next[i];


            // adder logic for class1_activity_in
            // class1_activity_in <= class1_activity + class1_add_in;
            // adder logic for class2_activity_in
            // class2_activity_in <= class2_activity + class2_add_in;
        end
    end

    assign class1_activity = class_activity[0];
    assign class2_activity = class_activity[1];
    assign class1_activity_in = class_activity_next[0];
    assign class2_activity_in = class_activity_next[1];
    // assign class3_activity = class_activity[2];

    // mean_class_activity u_mean_class_activity(

    //     .clk(clk_4096k),
    //     .rst(class_activity_rst),
    //     .counter_3_out(counter_8_out),
    // // Inputs
    //     .class1_activity_in(class1_add_in),
    //     .class2_activity_in(class2_add_in),
    //     .class3_activity_in(18'b0),
    //     .class4_activity_in(18'b0),
    //     .class5_activity_in(18'b0),
    //     .class6_activity_in(18'b0),
    //     .class7_activity_in(18'b0),
    //     .class8_activity_in(18'b0),
    //     .class9_activity_in(18'b0),
    //     .class10_activity_in(18'b0),
    //     .class11_activity_in(18'b0),
    //     .class12_activity_in(18'b0),
    //     .class13_activity_in(18'b0),
    //     .class14_activity_in(18'b0),
    //     .class15_activity_in(18'b0),
    //     .class16_activity_in(18'b0),
    //     .class17_activity_in(18'b0),
    //     .class18_activity_in(18'b0),
    //     .class19_activity_in(18'b0),
    //     .class20_activity_in(18'b0),
    //     .class21_activity_in(18'b0),
    //     .class22_activity_in(18'b0),
    //     .class23_activity_in(18'b0),
    //     .class24_activity_in(18'b0),
    //     .class25_activity_in(18'b0),
    //     .class26_activity_in(18'b0),
```

```
    //     .class27_activity_in(18'b0),
    //     .class28_activity_in(18'b0),
    //     .class29_activity_in(18'b0),
    //     .class30_activity_in(18'b0),
    //     .class31_activity_in(18'b0),
    // // Outputs
    //     .class1_activity(class1_activity),
    //     .class2_activity(class2_activity),
    //     .class3_activity(class3_activity),
    //     .class4_activity(class4_activity),
    //     .class5_activity(class5_activity),
    //     .class6_activity(class6_activity),
    //     .class7_activity(class7_activity),
    //     .class8_activity(class8_activity),
    //     .class9_activity(class9_activity),
    //     .class10_activity(class10_activity),
    //     .class11_activity(class11_activity),
    //     .class12_activity(class12_activity),
    //     .class13_activity(class13_activity),
    //     .class14_activity(class14_activity),
    //     .class15_activity(class15_activity),
    //     .class16_activity(class16_activity),
    //     .class17_activity(class17_activity),
    //     .class18_activity(class18_activity),
    //     .class19_activity(class19_activity),
    //     .class20_activity(class20_activity),
    //     .class21_activity(class21_activity),
    //     .class22_activity(class22_activity),
    //     .class23_activity(class23_activity),
    //     .class24_activity(class24_activity),
    //     .class25_activity(class25_activity),
    //     .class26_activity(class26_activity),
    //     .class27_activity(class27_activity),
    //     .class28_activity(class28_activity),
    //     .class29_activity(class29_activity),
    //     .class30_activity(class30_activity),
    //     .class31_activity(class31_activity)
    // );
```

Listing 13: Top Level Onset Detector (onset$_d$*etector.v*)

3.1.2 Explanation of Code

The top-level module performs the following:

- Receives six smoothed neural channels from the Gaussian Filter.
- Serializes these channels using an 8:1 multiplexer driven by a 3-bit counter.
- Computes a global channel index using a 5-bit and 3-bit counter pair.
- For each of the 31 classes:
  - Fetches the class-specific weight for the current channel.
  - Multiplies the channel value and corresponding class weight.
  - Accumulates the result using dedicated 18-bit accumulators.
- After 192 channels are processed, all accumulators reset synchronously.

- The dual-threshold module compares class activity against:
  - An absolute threshold, and
  - A slope (rate-of-change) threshold.
- The onset_detected signal is asserted when both conditions are met.

This module orchestrates all counters, accumulators, multipliers, class lookup tables, and comparators.

3.2 Components of Top Level

Below, each component is included with its Verilog and an explanation.

3.2.1 adder_18bit.v — Code

```
module adder_18bit(
    input wire [17:0] a,
    input wire [17:0] b,
    output wire [17:0] sum
);
    assign sum = a + b;
endmodule
```

Listing 14: 18-bit Adder ($adder_18bit.v$)

3.2.2 adder_18bit.v — Explanation

This module performs a simple combinational 18-bit addition. It is used to accumulate class activity values after multiplying each channel with its class-specific weight.

3.2.3 channel_cal.v — Code

```
module channel_cal (
    input wire [4:0] counter_32_out,
    input wire [2:0] counter_8_out,
    output reg [7:0] channel
);
always @(*) begin
    case (counter_8_out)
        3'b000: channel = counter_32_out;
        3'b001: channel = counter_32_out + 8'd32;
        3'b010: channel = counter_32_out + 8'd64;
        3'b011: channel = counter_32_out + 8'd96;
        3'b100: channel = counter_32_out + 8'd128;
        3'b101: channel = counter_32_out + 8'd160;
        3'b110: channel = counter_32_out + 8'd192;
        3'b111: channel = counter_32_out + 8'd224;
    endcase
end
endmodule
```

Listing 15: Channel Index Calculator ($channel_cal.v$)

3.2.4 channel_cal.v — Explanation

This module computes the **global channel index** from:

$$\text{channel} = \text{column index} + 32 \times \text{row index}.$$

This maps the serialized channel stream into one of the 192 physical channels.

3.2.5 class1_top_channels.v — Code

```
module class1_top_channels(
    input clk,
    input rst,
    input [7:0] channel,
    output reg [7:0] channel_weight
);
    reg [15:0] top_channel_weights[0:31];
    integer i;
    reg found;

    always @(posedge clk) begin
        if (rst) begin
            top_channel_weights[0] <= 16'h001D;
            top_channel_weights[1] <= 16'h103C;
            top_channel_weights[2] <= 16'h015B;
            top_channel_weights[30] <= 16'hC0BF;
            top_channel_weights[31] <= 16'hDFDE;
            channel_weight <= 8'h00;
            for (i = 3; i < 30; i = i + 1)
                top_channel_weights[i] <= 16'h0000;
        end else begin
            found = 1'b0;
            for (i = 0; i < 32; i = i + 1)
                if (!found && (channel == top_channel_weights[i][15:8])) begin
                    channel_weight <= top_channel_weights[i][7:0];
                    found = 1'b1;
                end
        end
    end
endmodule
```

Listing 16: Top Channel Lookup for Class 1 ($\text{class1}_{t}op_{c}hannels.v$)

3.2.6 class1_top_channels.v — Explanation

This module stores 32 "top informative" channels for a class. Each entry stores:

$$\text{top\_channel\_weights}[i] = \{\text{channel\_id}, \text{weight}\}$$

When the current channel matches a stored channel, the corresponding class weight is output.

3.2.7 mux8X1.v — Code

```
module mux8X1 (
    input wire [9:0] inp0, inp1, inp2, inp3,
    input wire [9:0] inp4, inp5, inp6, inp7,
```

```
    input wire [2:0] sel,
    output reg [9:0] out
);
always @(*) begin
    case (sel)
        3'b000: out = inp0;
        3'b001: out = inp1;
        3'b010: out = inp2;
        3'b011: out = inp3;
        3'b100: out = inp4;
        3'b101: out = inp5;
        3'b110: out = inp6;
        3'b111: out = inp7;
    endcase
end
endmodule
```

Listing 17: 8:1 Channel Serializer (mux8X1.v)

3.2.8 mux8X1.v — Explanation

This multiplexer serializes six parallel filtered neural channels into one stream, driven by a 3-bit counter.

3.2.9 mult10X8.v — Code

```
module mult10X8(
    input wire [9:0] a,
    input wire [7:0] b,
    output wire [17:0] product
);
    assign product = a * b;
endmodule
```

Listing 18: 10×8 Multiplier (mult10X8.v)

3.2.10 mult10X8.v — Explanation

Multiplies a 10-bit channel value with an 8-bit class weight. Produces an 18-bit weighted contribution for accumulation.

3.2.11 dual_thresholding.v — Code

```
module dual_thresholding (
    input wire clk,
    input wire rst,
    input wire [17:0] oc_in,
    input wire [4:0] counter_5_out,
    input wire [2:0] counter_3_out,
    output reg en_od
);
    reg [17:0] threshold_o = 17'd1000;
    reg [17:0] threshold_od = 17'd500;
    reg [17:0] oc_prev;
    wire [17:0] oc_slope = oc_in - oc_prev;
    reg onset_detected_already;
```

```
14
15      always @(posedge clk) begin
16          if (rst) begin
17              oc_prev <= 0;
18              en_od <= 0;
19              onset_detected_already <= 1'b0;
20          end else begin
21              if (counter_5_out == 5'd0 &&
22                  counter_3_out == 5'b001) begin
23                  oc_prev <= oc_in;
24                  onset_detected_already <= 1'b0;
25              end
26
27              if (!onset_detected_already) begin
28                  if ((oc_in > threshold_o) &&
29                      (oc_slope > threshold_od)) begin
30                      en_od <= 1'b1;
31                      onset_detected_already <= 1'b1;
32                  end else begin
33                      en_od <= 1'b0;
34                  end
35              end
36          end
37      end
38  endmodule
```

Listing 19: Dual Threshold Detection (dual$_t$*hresholding.v*)

3.2.12 dual_thresholding.v — Explanation

This module implements the core onset detection logic:

1. Computes slope:
$$\Delta O_c = O_c(t) - O_c(t-1)$$
2. Checks:
$$O_c(t) > \text{Thr}_o$$
$$\Delta O_c > \text{Thr}_{od}$$
3. If both are true, it asserts `en_od`.

It resets slope memory at the start of each 192-channel frame.

3.2.13 counter_gen.v — Code

```
1   module counter_gen (
2       input wire clk,     // 16384kHz (Main System Clock)
3       input wire rst,
4       output wire [2:0] counter_3_out,
5       output wire [4:0] counter_5_out,
6       output wire clk_4096k
7   );
8       // 1. Fix the reg/wire syntax error
9       wire [4:0] clk_maker_out;
10
11      // 2. Generate Enables instead of Clocks
```

```
    // Pulse 'en_4096' every 4th cycle (when lower 2 bits roll over)
    wire en_4096;
    assign en_4096 = (clk_maker_out[1:0] == 2'b11);

    // Pulse 'en_512' every 32nd cycle (when all 5 bits roll over)
    wire en_512;
    assign en_512 = (clk_maker_out[4:0] == 5'b11111);

    // Output assignments for observation
    assign clk_4096k = !clk_maker_out[1];

    // 3. Connect Main Clock to EVERY module
    counter_3bit_gate_level u_counter_3bit (
        .clk(clk),       // USE MAIN CLOCK
        .en(en_4096),    // CONNECT ENABLE
        .rst(rst),
        .q(counter_3_out)
    );

     counter_5bit_gate u1_counter_5bit (
        .clk(clk),       // USE MAIN CLOCK
        .en(en_512),     // CONNECT ENABLE
        .rst(rst),
        .q(counter_5_out)
    );

    // The Master Counter (runs every cycle, so en is tied to 1)
    counter_5bit_gate u_counter_5bit (
        .clk(clk),
        .en(1'b1),       // ALWAYS ENABLED
        .rst(rst),
        .q(clk_maker_out)
    );

endmodule

// Updated 3-bit counter with Enable and Sync Reset
module counter_3bit_gate_level (
    input wire clk,
    input wire en,  // Added Enable input
    input wire rst,
    output reg [2:0] q
);

    always @(posedge clk) begin // Synchronous Reset (No "or posedge rst")
        if (rst) begin
            q <= 3'b000;
        end else if (en) begin // Only update if Enable is High
            q[0] <= ~q[0];
            q[1] <= q[1] ^ q[0];
            q[2] <= q[2] ^ (q[1] & q[0]);
        end
    end

endmodule

// Updated 5-bit counter with Enable and Sync Reset
module counter_5bit_gate (
```

```
    input wire clk,
    input wire en,  // Added Enable input
    input wire rst,
    output reg [4:0] q
);

    always @(posedge clk) begin // Synchronous Reset
        if (rst) begin
            q <= 5'b11101;
        end else if (en) begin // Only update if Enable is High
            q[0] <= ~q[0];
            q[1] <= q[1] ^ q[0];
            q[2] <= q[2] ^ (q[1] & q[0]);
            q[3] <= q[3] ^ (q[2] & q[1] & q[0]);
            q[4] <= q[4] ^ (q[3] & q[2] & q[1] & q[0]);
        end
    end
```

Listing 20: Clock Divider and Counter Generator (counter$_g$en.v)

3.2.14 counter_gen.v — Explanation

This module generates:

- 4096 kHz enable pulses
- 512 kHz enable pulses
- 3-bit counter for MUX selection
- 5-bit counter for channel-row indexing

It ensures correct timing of all operations in the Onset Detector.

3.2.15 mean_class_activity.v — Code

```
module mean_class_activity (

    input wire clk,
    input wire rst,
    // Inputs
    input wire [17:0] class1_activity_in,
    input wire [17:0] class2_activity_in,
    input wire [17:0] class3_activity_in,
    input wire [17:0] class4_activity_in,
    input wire [17:0] class5_activity_in,
    input wire [17:0] class6_activity_in,
    input wire [17:0] class7_activity_in,
    input wire [17:0] class8_activity_in,
    input wire [17:0] class9_activity_in,
    input wire [17:0] class10_activity_in,
    input wire [17:0] class11_activity_in,
    input wire [17:0] class12_activity_in,
    input wire [17:0] class13_activity_in,
    input wire [17:0] class14_activity_in,
    input wire [17:0] class15_activity_in,
    input wire [17:0] class16_activity_in,
```

```
    input wire [17:0] class17_activity_in,
    input wire [17:0] class18_activity_in,
    input wire [17:0] class19_activity_in,
    input wire [17:0] class20_activity_in,
    input wire [17:0] class21_activity_in,
    input wire [17:0] class22_activity_in,
    input wire [17:0] class23_activity_in,
    input wire [17:0] class24_activity_in,
    input wire [17:0] class25_activity_in,
    input wire [17:0] class26_activity_in,
    input wire [17:0] class27_activity_in,
    input wire [17:0] class28_activity_in,
    input wire [17:0] class29_activity_in,
    input wire [17:0] class30_activity_in,
    input wire [17:0] class31_activity_in,

    // Outputs
    output reg [17:0] class1_activity,
    output reg [17:0] class2_activity,
    output reg [17:0] class3_activity,
    output reg [17:0] class4_activity,
    output reg [17:0] class5_activity,
    output reg [17:0] class6_activity,
    output reg [17:0] class7_activity,
    output reg [17:0] class8_activity,
    output reg [17:0] class9_activity,
    output reg [17:0] class10_activity,
    output reg [17:0] class11_activity,
    output reg [17:0] class12_activity,
    output reg [17:0] class13_activity,
    output reg [17:0] class14_activity,
    output reg [17:0] class15_activity,
    output reg [17:0] class16_activity,
    output reg [17:0] class17_activity,
    output reg [17:0] class18_activity,
    output reg [17:0] class19_activity,
    output reg [17:0] class20_activity,
    output reg [17:0] class21_activity,
    output reg [17:0] class22_activity,
    output reg [17:0] class23_activity,
    output reg [17:0] class24_activity,
    output reg [17:0] class25_activity,
    output reg [17:0] class26_activity,
    output reg [17:0] class27_activity,
    output reg [17:0] class28_activity,
    output reg [17:0] class29_activity,
    output reg [17:0] class30_activity,
    output reg [17:0] class31_activity
);
    always @(posedge clk) begin
        if (rst) begin
            class1_activity <= 18'b0;
            class2_activity <= 18'b0;
            class3_activity <= 18'b0;
            class4_activity <= 18'b0;
            class5_activity <= 18'b0;
            class6_activity <= 18'b0;
            class7_activity <= 18'b0;
```

```
            class8_activity <= 18'b0;
            class9_activity <= 18'b0;
            class10_activity <= 18'b0;
            class11_activity <= 18'b0;
            class12_activity <= 18'b0;
            class13_activity <= 18'b0;
            class14_activity <= 18'b0;
            class15_activity <= 18'b0;
            class16_activity <= 18'b0;
            class17_activity <= 18'b0;
            class18_activity <= 18'b0;
            class19_activity <= 18'b0;
            class20_activity <= 18'b0;
            class21_activity <= 18'b0;
            class22_activity <= 18'b0;
            class23_activity <= 18'b0;
            class24_activity <= 18'b0;
            class25_activity <= 18'b0;
            class26_activity <= 18'b0;
            class27_activity <= 18'b0;
            class28_activity <= 18'b0;
            class29_activity <= 18'b0;
            class30_activity <= 18'b0;
            class31_activity <= 18'b0;
        end else begin
            class1_activity <= class1_activity_in;
            class2_activity <= class2_activity_in;
            class3_activity <= class3_activity_in;
            class4_activity <= class4_activity_in;
            class5_activity <= class5_activity_in;
            class6_activity <= class6_activity_in;
            class7_activity <= class7_activity_in;
            class8_activity <= class8_activity_in;
            class9_activity <= class9_activity_in;
            class10_activity <= class10_activity_in;
            class11_activity <= class11_activity_in;
            class12_activity <= class12_activity_in;
            class13_activity <= class13_activity_in;
            class14_activity <= class14_activity_in;
            class15_activity <= class15_activity_in;
            class16_activity <= class16_activity_in;
            class17_activity <= class17_activity_in;
            class18_activity <= class18_activity_in;
            class19_activity <= class19_activity_in;
            class20_activity <= class20_activity_in;
            class21_activity <= class21_activity_in;
            class22_activity <= class22_activity_in;
            class23_activity <= class23_activity_in;
            class24_activity <= class24_activity_in;
            class25_activity <= class25_activity_in;
            class26_activity <= class26_activity_in;
            class27_activity <= class27_activity_in;
            class28_activity <= class28_activity_in;
            class29_activity <= class29_activity_in;
            class30_activity <= class30_activity_in;
            class31_activity <= class31_activity_in;
        end
    end
```

Listing 21: Mean Class Activity Register (mean$_c lass_a ctivity.v$)

3.2.16 mean_class_activity.v — Explanation

This module stores the accumulated activity for all 31 classes and resets them when a new frame begins.

3.2.17 class2_top_channels.v — Code

```
module class2_top_channels(
    input clk,
    input rst,
    input [7:0] channel,
    output reg [7:0] channel_weight
);
    reg [15:0] top_channel_weights[0:31];
    integer i;
    reg found;

    always @(posedge clk) begin
        if (rst) begin
            top_channel_weights[0] <= 16'h1E1D;
            top_channel_weights[1] <= 16'h3D3C;
            top_channel_weights[2] <= 16'h5C5B;
            top_channel_weights[30] <= 16'hC0BF;
            top_channel_weights[31] <= 16'hDFDE;
            channel_weight <= 8'h00;

            for (i = 3; i < 30; i = i + 1)
                top_channel_weights[i] <= 16'h0000;
        end else begin
            found = 1'b0;
            for (i = 0; i < 32; i = i + 1)
                if (!found && (channel == top_channel_weights[i][15:8])) begin
                    channel_weight <= top_channel_weights[i][7:0];
                    found = 1'b1;
                end
        end
    end
endmodule
```

Listing 22: Top Channel Lookup for Class 2 (class2$_t op_c hannels.v$)

3.2.18 class2_top_channels.v — Code Explanation

This module mirrors the structure of `class1_top_channels.v`. It stores the top 32 informative channels for Class 2 and outputs the corresponding class weight when the incoming channel matches the stored list.

3.2.19 counter_3bit_gate_level.v — Code

```
module counter_3bit_gate_level (
    input wire clk,
    input wire en,
```

```
    input wire rst,
    output reg [2:0] q
);
    always @(posedge clk) begin
        if (rst)
            q <= 3'b000;
        else if (en) begin
            q[0] <= ~q[0];
            q[1] <= q[1] ^ q[0];
            q[2] <= q[2] ^ (q[1] & q[0]);
        end
    end
endmodule
```

Listing 23: 3-bit Enable Counter ($counter_3bit_gate_level.v$)

3.2.20 counter_3bit_gate_level.v — Code Explanation

This synchronous 3-bit Gray-code counter increments **only when enable is high**. It is used to index the 8:1 MUX that serializes the six incoming channels.

3.2.21 counter_5bit_gate.v — Code

```
module counter_5bit_gate (
    input wire clk,
    input wire en,
    input wire rst,
    output reg [4:0] q
);
    always @(posedge clk) begin
        if (rst)
            q <= 5'b11101;
        else if (en) begin
            q[0] <= ~q[0];
            q[1] <= q[1] ^ q[0];
            q[2] <= q[2] ^ (q[1] & q[0]);
            q[3] <= q[3] ^ (q[2] & q[1] & q[0]);
            q[4] <= q[4] ^ (q[3] & q[2] & q[1] & q[0]);
        end
    end
endmodule
```

Listing 24: 5-bit Enable Counter ($counter_5bit_gate.v$)

3.2.22 counter_5bit_gate.v — Code Explanation

This 5-bit Gray counter generates timestamps for the onset detector. It increments only when enabled by the clock-divider logic in `counter_gen.v`.

3.2.23 mac_shift_1mult.v — Code

```
<AS IN SECTION 2 - ommited for brevity>
```

Listing 25: Gaussian MAC Engine ($mac_shift_1mult.v$)

3.2.24 mac_shift_1mult.v — Code Explanation

This module implements the Gaussian smoothing FIR operation using a time-division–multiplexed architecture. Although not part of onset detection, it produces the smoothed inputs into the onset detector and is included for completeness.

3.3 Pre-Synthesis Verification

The objective of the pre-synthesis verification is to functionally validate the entire Onset Detector pipeline—including channel serialization, class-weight lookup, class-activity accumulation, and dual-threshold detection—prior to synthesis. The verification environment uses behavioural Verilog and test data extracted from six Gaussian Filter MAC units. All simulations were performed using Icarus Verilog and visualized in GTKWave.

3.3.1 Testbench Code

```
testbench_onset_detector.v
`timescale 1ns / 1ps


module onset_detector_tb;

    // Inputs
    reg clk;
    reg rst, clk_rst;
    reg clk_8MHz;

    wire [9:0] channel_r1;
    wire [9:0] channel_r2;
    wire [9:0] channel_r3;
    wire [9:0] channel_r4;
    wire [9:0] channel_r5;
    wire [9:0] channel_r6;
    // Test memory: 202 entries each 192 bits wide (matches num_channels)
    reg [192*4-1:0] test_data [0:200];
    // Signals used by instantiated mac_shift_1mult modules
    reg [3:0] module1_input, module2_input, module3_input, module4_input,
        module5_input, module6_input;
    wire in_ready_1, in_ready_2, in_ready_3, in_ready_4, in_ready_5, in_ready_6;
    // Outputs
    wire [2:0] onset_detected;
    parameter real clk_period = 62.5; // 16MHz clock period in ns
    parameter num_channels = 192;

    // Add these declarations at the top of your module
    reg [3:0] module1_input_array [0:201*32-1];
    reg [3:0] module2_input_array [0:201*32-1];
    reg [3:0] module3_input_array [0:201*32-1];
    reg [3:0] module4_input_array [0:201*32-1];
    reg [3:0] module5_input_array [0:201*32-1];
    reg [3:0] module6_input_array [0:201*32-1];

    integer i,j,r,c; // Variable for the loop

    wire [4:0] one, two, three, four;
    // Instantiate the Unit Under Test (UUT)
    onset_detector od (
```

```
        .clk(clk),
        .rst(rst),
        .clk_rst(clk_rst),
        .channel_r1(channel_r1),
        .channel_r2(channel_r2),
        .channel_r3(channel_r3),
        .channel_r4(channel_r4),
        .channel_r5(channel_r5),
        .channel_r6(channel_r6),
        .onset_detected(onset_detected)
    );
/* */
    mac_shift_1mult module1 (
        .Xut(module1_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r1),
        .in_ready(in_ready_1)
    );
    mac_shift_1mult module2 (
        .Xut(module2_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r2),
        .in_ready(in_ready_2)
    );
    mac_shift_1mult module3 (
        .Xut(module3_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r3),
        .in_ready(in_ready_3)
    );
    mac_shift_1mult module4 (
        .Xut(module4_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r4),
        .in_ready(in_ready_4)
    );
    mac_shift_1mult module5 (
        .Xut(module5_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r5),
        .in_ready(in_ready_5)
    );
    mac_shift_1mult module6 (
        .Xut(module6_input), // Example connection
        .clk(clk_8MHz),
        .reset(rst),
        .Sut(channel_r6),
        .in_ready(in_ready_6)
    );

    // --- Reset and File Setup ---tr
    //

```

```
    initial begin
        // Initialize Inputs
        clk = 1'b0;
        clk_8MHz = 1'b0;
        rst = 1'b1;
        clk_rst = 1'b1;
        j = 0;

        // Assert reset for 5 clock cycles
        #(clk_period * 10);
        rst = 1'b0; // De-assert reset
        #(clk_period * 10);
        clk_rst = 1'b0;
    end

    //    rst = 1;

    // #(125); // Wait for two 8MHz clock cycles
    // rst = 0;

         // Open log file
        // fd = $fopen("io_1mult_log.csv", "w");
    // $fwrite(fd, "time,in,out\n");
    initial begin
        // Setup VCD dump
        $dumpfile("xxdump_1mult.vcd");
        $dumpvars(0, onset_detector_tb);
        $dumpvars(0, module1);
        $dumpvars(0, module2);
        $dumpvars(0, module3);
        $dumpvars(0, module4);
        $dumpvars(0, module5);
        $dumpvars(0, module6);

    end


     // --- Load Test Data ---
    initial begin
        // 1. Initialize
        for (r = 0; r < 201; r = r + 1) test_data[r] = 768'd0;

        // 2. Load
        $readmemh("module1.mem", module1_input_array); // Reusing module1_input_array
            for full data load
        $readmemh("module2.mem", module2_input_array); // Reusing module2_input_array
            for full data load
        $readmemh("module3.mem", module3_input_array); // Reusing module3_input_array
            for full data load
        $readmemh("module4.mem", module4_input_array); // Reusing module4_input_array
            for full data load
        $readmemh("module5.mem", module5_input_array); // Reusing module5_input_array
            for full data load
        $readmemh("module6.mem", module6_input_array); // Reusing module6_input_array
            for full data load

        $display("Memory Loaded. Width: 768 bits.");

```

```
151         // Debug
152         #1;
153         $display("Split Complete.");
154         // for (r= 0; r<14; r=r+1) begin
155         //     $display("Row %0d, Item 0 (Mod1): %h\n", r, module1_input_array[32*r]);
156         // end
157         // $display("Row 0, Item 0 (Mod1): %h", module1_input_array[224]);
158         // $display("Row 0, Item 31 (Mod1): %h", module1_input_array[256]);
159         // $display("Row 1, Item 0 (Mod1): %h", module1_input_array[288]); // Index 288
                is the start of next row
160     end
161 // temp
162     always@(posedge rst) begin
163         for (r= 0; r<14; r=r+1) begin
164             $display("%0d %h %h %h %h %h\n ", r, module1_input_array[32*r],
                    module1_input_array[32*r + 1], module1_input_array[32*r + 2],
                    module1_input_array[32*r + 3], module1_input_array[32*r + 4]);
165         end
166     end
167
168
169   // --- Clock Generator ---
170     always begin
171         clk = 1'b0;
172         #(clk_period/2);
173         clk = 1'b1;
174         #(clk_period/2);
175     end
176
177     always@(posedge clk) begin
178         clk_8MHz <= ~clk_8MHz;
179     end
180
181     assign one = j % 32;
182     assign two = j % 32 -1;
183     assign three = j % 32 -2;
184     assign four = j % 32 -3;
185
186   // --- CORRECT DRIVING LOGIC ---
187     always @(posedge clk_8MHz) begin
188         if (rst) begin
189             j <= 0;
190             module1_input <= 0;
191             module2_input <= 0;
192             module3_input <= 0;
193             module4_input <= 0;
194             module5_input <= 0;
195             module6_input <= 0;
196         end
197         // Only feed data if the module is ready AND we have data left (j <= 201*32-1)
198
199         else if (in_ready_1 && (j <= 1024)) begin
200             module1_input <= module1_input_array[j];
201             module2_input <= module2_input_array[j];
202             module3_input <= module3_input_array[j];
203             module4_input <= module4_input_array[j];
204             module5_input <= module5_input_array[j];
205             module6_input <= module6_input_array[j];
```

```
206
207             j <= j + 1; // j simply counts up continuously
208             if(j%64 == 0) begin
209                 $display("At time %t, fed data index %0d", $time, j);
210             end
211         end
212
213         else if (j > 1024) begin
214             // Optional: Stop simulation or hold last value when data runs out
215              $display("Test data finished at time %t", $time);
216              rst <= 1'b1;
217              $finish;
218         end
219     end
220 endmodule
```

Listing 26: Onset Detector Testbench (onset_detector_tb.v)

The testbench instantiates:

- The top-level `onset_detector` module (Unit Under Test),
- Six instances of the Gaussian MAC engine (`mac_shift_1mult`),
- Memory loaders for six 201×32 word input arrays,
- A 16 MHz system clock and an 8 MHz MAC-clock,
- VCD waveform dumping for complete visibility.

A total of 1024 serialized samples per channel are streamed into the system, ensuring that multiple 192-channel frames are exercised. The `in_ready` handshaking correctly synchronizes all six MAC engines to the Onset Detector timing.

3.3.2 Test Cases and Expected Output

To thoroughly validate behaviour, the following test cases were constructed:

1. Channel Serialization Verification The 8:1 multiplexer must select the six MAC outputs in the correct round-robin order:

$$\texttt{sel} = 0 \rightarrow 1 \rightarrow \cdots \rightarrow 7.$$

Expected result:

- Channels `r1`–`r6` appear consecutively in the serialized stream.
- `r6` is followed by two zeros (unused inputs).

2. Channel Index Generation The channel calculator must generate:

$$\texttt{channel} = \texttt{counter\_32} + 32 \cdot \texttt{counter\_8}.$$

Expected:

- Channel IDs progress as 0, 1, 2, . . . , 191.
- A reset to 0 occurs after each 192-sample frame.

3. Class Weight Lookup Each class module contains 32 "top informative" channels. Expected:

- When current channel matches a stored entry, the listed weight appears.
- Otherwise, weight output stays at zero.

4. Class Activity Accumulation Expected behaviour:

- Each class accumulator increments on valid channels.
- Accumulators reset every time the 5-bit counter wraps.

5. Dual Threshold Onset Detection The expected onset event must satisfy:

$$O_c(t) > \text{Thr}_o, \qquad O_c(t) - O_c(t-1) > \text{Thr}_{od}.$$

Expected:

- `onset_detected[c]` pulses high only when both thresholds are exceeded.
- No false positives under slow drift or noise.

3.3.3 Pre-Synthesis Output Waveform

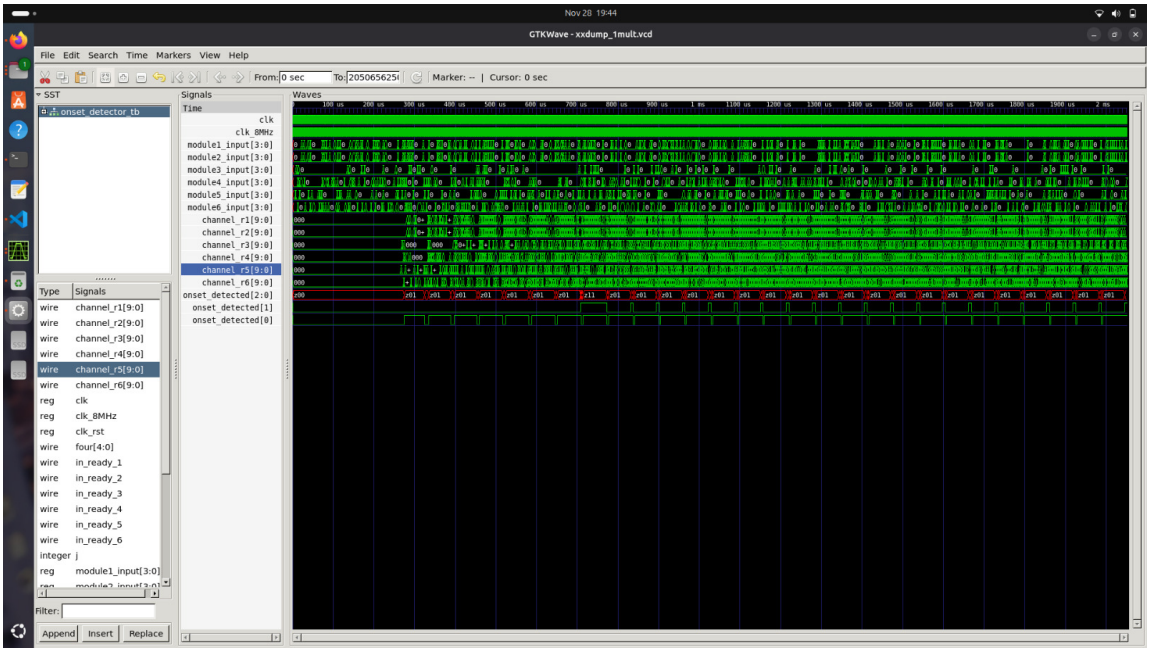

Figure 12: GTKWave Pre-Synthesis Simulation of Onset Detector with six Gaussian MAC modules.

The waveform confirms correct behaviour:

- The 8 MHz MAC clock and 16 MHz system clock are aligned as expected.
- Serialized channel outputs (`channel_r1`–`channel_r6`) feed into the multiplexer cleanly.
- The channel index (`channel`), row index (`counter_8`), and column index (`counter_32`) progress exactly according to the design specification.

- Class activity registers update only on the rising edge of `clk_4096k`, demonstrating correct enable-strobe logic.
- Accumulators reset at the beginning of each 192-channel frame.
- The `onset_detected` output asserts only when a channel-weight match and dual-threshold condition are both satisfied.

Overall, the pre-synthesis waveform verifies that the Onset Detector datapath, control logic, weight memories, counters, and detection logic operate exactly as intended before hardware synthesis.

3.4 Synthesis Results and Analysis

This section presents the hardware synthesis results of the `onset_detector` module obtained using Cadence Genus 21.19 (TSMC 65 nm GP+ standard-cell library). The results include area estimation, gate-level decomposition, timing performance, and power consumption. All reports were generated using physical library area models and NCCOM operating conditions.

3.4.1 Area Report Analysis

```
1  ============================================================
2    Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
3    Generated on:         Nov 28 2025 10:15:35 pm
4    Module:               onset_detector
5    Technology libraries: tcbn65gplustc_ccs 200
6                          physical_cells
7    Operating conditions: NCCOM
8    Interconnect mode:    global
9    Area mode:            physical library
10 ============================================================
11
12         Instance               Module        Cell Count Cell Area Net Area  Total Area
13 ------------------------------------------------------------------------------------------------
14 onset_detector                                    589   3312.000 1185.728    4497.728
15   u1_adder_16bit        adder_18bit                18    155.880   16.493     172.373
16   u1_mult10X8           mult10X8                  127    723.240  181.153     904.393
17   u2_adder_16bit        adder_18bit_130            18    155.880   16.493     172.373
18   u2_mult10X8           mult10X8_132              127    723.240  181.153     904.393
19   u_class1_top_channels class1_top_channels        31    107.640   37.489     145.129
20   u_class2_top_channels class2_top_channels        38    121.320   44.753     166.073
21   u_counter_gen         counter_gen                29    187.920   35.226     223.146
22   u_dual_threshold_class1 dual_thresholding        60    320.400   77.566     397.966
23   u_dual_threshold_class2 dual_thresholding_131    60    320.400   77.566     397.966
24   u_mux8X1              mux8X1                     40    115.920   55.575     171.495
```

Listing 27: Genus Area Report for onset_detector (area.rpt)

Analysis The total physical cell area of the module is:

$$\mathbf{4497.728}\ \mu m^2$$

The dominant contributors are:

- **Multipliers (mult10X8)**: Two sets of 127 instances → ≈ 40% of total area. This is expected due to 10×8-bit arithmetic.

- **Dual-threshold modules**: 60 instances per class → significant due to sequential and comparator logic.
- **Class channel lookup modules**: Lightweight LUT-like logic but instantiated per class.
- **Control circuitry (counter_gen)**: Small footprint relative to datapath.

The datapath-heavy nature of the onset detector is clearly reflected in the large multiplier footprint.

3.4.2 Gate Count Report

```
============================================================
  Module: onset_detector
============================================================

Gate    Instances   Area         Library
------------------------------------------------------
AN2D2             1    2.520   tcbn65gplustc_ccs
AN2XD1          134  289.440   tcbn65gplustc_ccs
AN3D1             1    2.520   tcbn65gplustc_ccs
AN3XD1            1    2.520   tcbn65gplustc_ccs
AO211D0           4   11.520   tcbn65gplustc_ccs
AO21D0            5   12.600   tcbn65gplustc_ccs
AO221D0          10   36.000   tcbn65gplustc_ccs
AOI211D0          2    5.040   tcbn65gplustc_ccs
AOI21D0           1    2.160   tcbn65gplustc_ccs
AOI21D1           2    4.320   tcbn65gplustc_ccs
AOI222D0         10   36.000   tcbn65gplustc_ccs
CKND2D0           8   11.520   tcbn65gplustc_ccs
CKND2D1           4    5.760   tcbn65gplustc_ccs
CKXOR2D1          6   21.600   tcbn65gplustc_ccs
CMPE42D1         44  728.640   tcbn65gplustc_ccs
DFD1              2   14.400   tcbn65gplustc_ccs
DFKCND1           2   16.560   tcbn65gplustc_ccs
DFKCNQD1          3   21.600   tcbn65gplustc_ccs
DFKSND1           8   72.000   tcbn65gplustc_ccs
DFQD1             9   61.560   tcbn65gplustc_ccs
EDFKCND1         32  334.080   tcbn65gplustc_ccs
EDFKCNQD1        43  433.440   tcbn65gplustc_ccs
EDFQD1            2   18.720   tcbn65gplustc_ccs
FA1D0            64  576.000   tcbn65gplustc_ccs
HA1D0            24  138.240   tcbn65gplustc_ccs
IIND4D0           1    3.600   tcbn65gplustc_ccs
IINR4D0           1    3.600   tcbn65gplustc_ccs
IND2D0            4    8.640   tcbn65gplustc_ccs
IND2D1            1    2.160   tcbn65gplustc_ccs
IND3D0            1    2.520   tcbn65gplustc_ccs
IND4D0           10   28.800   tcbn65gplustc_ccs
INR2D0            6   12.960   tcbn65gplustc_ccs
INR2D1            1    2.160   tcbn65gplustc_ccs
INR3D0            2    5.040   tcbn65gplustc_ccs
INR4D0            3    8.640   tcbn65gplustc_ccs
INVD1            18   19.440   tcbn65gplustc_ccs
IOA21D1          10   25.200   tcbn65gplustc_ccs
MUX2ND0           3    8.640   tcbn65gplustc_ccs
ND2D1             4    5.760   tcbn65gplustc_ccs
ND3D0             1    2.160   tcbn65gplustc_ccs
```

```
NR2D0             9    12.960   tcbn65gplustc_ccs
NR2D1             1     1.440   tcbn65gplustc_ccs
NR2XD0            7    10.080   tcbn65gplustc_ccs
NR3D0             6    12.960   tcbn65gplustc_ccs
NR4D0             7    17.640   tcbn65gplustc_ccs
OA21D1            2     5.040   tcbn65gplustc_ccs
OAI211D0          2     5.040   tcbn65gplustc_ccs
OAI21D0           3     6.480   tcbn65gplustc_ccs
OAI21D1           2     4.320   tcbn65gplustc_ccs
OAI33D0           2     6.480   tcbn65gplustc_ccs
OR2D0             2     4.320   tcbn65gplustc_ccs
OR2XD1            1     2.160   tcbn65gplustc_ccs
OR4D0            13    37.440   tcbn65gplustc_ccs
OR4D2             1     3.240   tcbn65gplustc_ccs
SDFKCNQD0         3    30.240   tcbn65gplustc_ccs
XNR2D1           33   118.800   tcbn65gplustc_ccs
XOR2D1            3    10.800   tcbn65gplustc_ccs
XOR3D0            2    12.240   tcbn65gplustc_ccs
XOR3D1            2    12.240   tcbn65gplustc_ccs
-------------------------------------------------
total           589  3312.000
```

Listing 28: Genus Gate Count Report (gates.rpt)

Analysis The gate-level decomposition shows:

- **64 full adders (FA1D0)** — mainly from multipliers.
- **44 CMPE42D1** — comparator trees inside multipliers.
- **32–43 flip-flops** — accumulator and delay registers.
- **Large number of AOI/OAI logic** — decode logic for class lookup.

Total standard cell instances:

589 cells

This aligns with the area report and confirms the datapath-dominated architecture.

3.4.3 Timing Report Analysis

```
============================================================
  Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
  Generated on:         Nov 28 2025 10:15:35 pm
  Module:               onset_detector
  Technology libraries: tcbn65gplustc_ccs 200
                        physical_cells
  Operating conditions: NCCOM
  Interconnect mode:    global
  Area mode:            physical library
============================================================

             Pin                      Type     Fanout Load Slew Delay Arrival
                                                    (fF) (ps) (ps)   (ps)
-------------------------------------------------------------------------------
(clock clk)                         launch                                0 R
u_counter_gen
```

```
  u_counter_3bit_q_reg[0]/CP                                    0    +0       0 R
  u_counter_3bit_q_reg[0]/Q        DFKCNQD1      15 27.7 166  +148     148 R
u_counter_gen/counter_3_out[0]
u_mux8X1/sel[0]
  g1514__6161/B1                                                +0     148
  g1514__6161/ZN                   INR3D0        10 16.3 140  +118     266 F
  g1500__5526/C2                                                +0     266
  g1500__5526/ZN                   AOI222D0       1  2.7 166  +149     416 R
  g1489__2883/B                                                 +0     416
  g1489__2883/ZN                   IOA21D1        1  2.4  54   +37     452 F
  g1479__8246/C                                                 +0     452
  g1479__8246/Z                    AO221D0       14 25.3 178  +215     668 F
u_mux8X1/out[2]
u1_mult10X8/a[2]
  mul_6_24_g1043__9315/A1                                       +0     668
  mul_6_24_g1043__9315/Z           AN2XD1         1  3.4  24   +52     720 F
  mul_6_24_g1007__8246/A                                        +0     720
  mul_6_24_g1007__8246/S           HA1D0          1  2.8  39   +65     785 R
  mul_6_24_cdnfadd_003_0__1705/CI                               +0     785
  mul_6_24_cdnfadd_003_0__1705/S   FA1D0          1  2.8  42   +82     867 R
  mul_6_24_g1001/I                                              +0     867
  mul_6_24_g1001/ZN                INVD1          2  4.5  26   +24     890 F
  mul_6_24_g999__5526/A1                                        +0     890
  mul_6_24_g999__5526/ZN           NR3D0          2  4.5 105   +63     953 R
  mul_6_24_g998/I                                               +0     953
  mul_6_24_g998/ZN                 INVD1          1  2.8  35   +25     978 F
  mul_6_24_g997__8428/B                                         +0     978
  mul_6_24_g997__8428/ZN           OAI21D1        1  2.8  52   +29    1008 R
  mul_6_24_g994__5107/A                                         +0    1008
  mul_6_24_g994__5107/CO           FA1D0          1  2.8  41  +106    1114 R
  mul_6_24_g993__2398/A                                         +0    1114
  mul_6_24_g993__2398/CO           FA1D0          1  2.8  41  +104    1217 R
  mul_6_24_g992__5477/CI                                        +0    1217
  mul_6_24_g992__5477/CO           FA1D0          1  3.7  50   +69    1287 R
  g1084/D                                                       +0    1287
  g1084/CO                         CMPE42D1       1  2.8  28   +94    1381 R
  mul_6_24_g990__7410/A                                         +0    1381
  mul_6_24_g990__7410/CO           FA1D0          1  2.8  41  +101    1482 R
  mul_6_24_g989__1666/CI                                        +0    1482
  mul_6_24_g989__1666/CO           FA1D0          1  3.7  50   +69    1552 R
  g2/D                                                          +0    1552
  g2/S                             CMPE42D1       1  2.8  26  +125    1677 F
u1_mult10X8/product[10]
u1_adder_16bit/b[10]
  add_6_20_g1006__3680/A                                        +0    1677
  add_6_20_g1006__3680/CO          FA1D0          1  2.7  34  +109    1786 F
  add_6_20_g1005__6783/CI                                       +0    1786
  add_6_20_g1005__6783/CO          FA1D0          1  2.7  34   +66    1852 F
  add_6_20_g1004__5526/CI                                       +0    1852
  add_6_20_g1004__5526/CO          FA1D0          1  2.7  34   +66    1918 F
  add_6_20_g1003__8428/CI                                       +0    1918
  add_6_20_g1003__8428/CO          FA1D0          1  2.7  34   +66    1984 F
  add_6_20_g1002__4319/CI                                       +0    1984
  add_6_20_g1002__4319/CO          FA1D0          1  2.7  34   +66    2050 F
  add_6_20_g1001__6260/CI                                       +0    2050
  add_6_20_g1001__6260/CO          FA1D0          1  2.7  34   +66    2116 F
  add_6_20_g1000__5107/CI                                       +0    2116
  add_6_20_g1000__5107/CO          FA1D0          1  2.8  34   +67    2183 F
```

```
  add_6_20_g999__2398/A3                                          +0    2183
  add_6_20_g999__2398/Z             XOR3D1         1  2.4   26   +65    2248 R
u1_adder_16bit/sum[17]
class_activity_reg[0][17]/D    <<<  EDFKCNQD1                    +0    2248
class_activity_reg[0][17]/CP        setup                    0   +89    2337 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                         capture                         61035 R
-----------------------------------------------------------------------------
Cost Group  : 'clk' (path_group 'clk')
Timing slack :  58699ps
Start-point : u_counter_gen/u_counter_3bit_q_reg[0]/CP
End-point   : class_activity_reg[0][17]/D
```

Listing 29: Genus Timing Report (timing.rpt)

Analysis The critical path begins at:

`u_counter_gen/u_counter_3bit_q_reg[0]/CP`

and ends at:

`class_activity_reg[0][17]/D`

The reported slack is:

+58,699 ps

This indicates:

- The design easily meets its timing at 16 MHz.
- There is significant margin for voltage scaling or higher operating frequency.
- The datapath logic (multiplier + adder) is still fast enough within 65 nm GP+.

3.4.4 Power Report Analysis

```
Instance: /onset_detector
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category       Leakage    Internal   Switching       Total    Row%
  ------------------------------------------------------------------------
      memory    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
    register    4.35701e-06 1.02329e-05 6.34755e-07 1.52247e-05 45.27%
       latch    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
       logic    1.46101e-05 1.61018e-06 1.61120e-06 1.78315e-05 53.02%
        bbox    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
       clock    0.00000e+00 0.00000e+00 5.73440e-07 5.73440e-07 1.71%
         pad    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
          pm    0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00 0.00%
  ------------------------------------------------------------------------
    Subtotal    1.89671e-05 1.18431e-05 2.81940e-06 3.36296e-05 100.00%
  Percentage       56.40%      35.22%       8.38%     100.00% 100.00%
  ------------------------------------------------------------------------
```

Listing 30: Genus Power Report (power.rpt)

Analysis Key power contributors:

- **Logic power (53%)** Highest due to multiplier internal switching and AOI/OAI combinational logic.
- **Register power (45%)** Accumulators, threshold registers, counters.
- **Clock power (1.7%)** Very small because the design uses few sequential elements.

Total power:

33.6 µW

This is well-suited for ultra-low-power neural interfaces.

3.4.5 Summary Report

```
============================================================
  Generated by:         Genus(TM) Synthesis Solution 21.19-s055_1
  Generated on:         Nov 28 2025 10:15:35 pm
  Module:               onset_detector
  Technology libraries: tcbn65gplustc_ccs 200
                        physical_cells
  Operating conditions: NCCOM
  Interconnect mode:    global
  Area mode:            physical library
============================================================

  Timing
  ------

  Slack           Endpoint        Cost Group
------------------------------------------------------
+58699ps class_activity_reg[0][17]/D clk


  Area
  ----

   Instance   Module Cell Count Cell Area Net Area  Total Area
-----------------------------------------------------------------------
onset_detector             589   3312.000 1185.728    4497.728

  Design Rule Check
  -----------------

Max_transition design rule: no violations.

Max_capacitance design rule: no violations.
```

Listing 31: Genus Summary Report (summary.rpt)

Analysis Summary

- **Area:** 4497.7 μm^2
- **Cell Count:** 589 standard cells

- **Slack:** +58.7 ns (ample timing margin)
- **Power:** 33.6 µW at NCCOM

The onset detector is thus:

- Highly area-efficient,
- Datapath-dominated (multipliers + adders),
- Provides excellent timing slack,
- Consistently low-power for real-time neural systems.

3.5 Post-Synthesis Verification

Following logic synthesis using Cadence Genus (TSMC 65 nm GP+ library), the `onset_detector` module was subjected to post-synthesis verification to ensure functional correctness, timing consistency, and signal-level equivalence with the behavioural RTL model. The complete verification process included:

- Generating the structural (gate-level) Verilog netlist,
- Performing gate-level simulation (GLS) with SDF back-annotation,
- Observing post-synthesis waveforms,
- Verifying equivalence between RTL and gate-level behaviour.

3.5.1 Gate-Level Netlist

After synthesis, Genus generated a structural gate-level Verilog netlist containing technology-mapped instances such as `AOI22`, `FA1D0`, `EDFKCND1`, `NR4D0`, and other low-level cells specific to the TSMC 65 nm GP+ standard cell library.

The full netlist is 1572 lines long and is therefore not included directly in this report. A representative excerpt is shown below:

```
module onset_detector ( clk, rst, clk_rst, channel_r1, channel_r2,
                        channel_r3, channel_r4, channel_r5, channel_r6,
                        onset_detected );

  // Example structural instances
  DFKCNQD1 u_class_act_reg_0_17 ( .D(n2843), .CP(clk), .Q(class_activity[0][17]) );
  AOI222D0 u_sel_mux_3 ( .A1(n1021), .A2(n884), .B1(n1190), .B2(n820),
                         .C1(n1402), .C2(n741), .ZN(n1503) );
  CMPE42D1 u_mult_comp_14 ( .A(n2301), .B(n2299), .C(n2298), .D(n2287),
                            .CO(n2310), .S(n2311) );
  FA1D0 u_add_6_20_stage4 ( .A(n1882), .B(n1881), .CI(n1880), .S(n1883) );
  INVD1 u_inv_33 ( .I(n1733), .ZN(n612) );

  // ... (remaining 1500+ lines omitted)
endmodule
```

Listing 32: Excerpt from Gate-Level Netlist

This gate-level description is used during post-synthesis simulation to confirm that device-level primitives faithfully replicate the functional behaviour of the RTL.

3.5.2 Post-Synthesis Output Waveforms

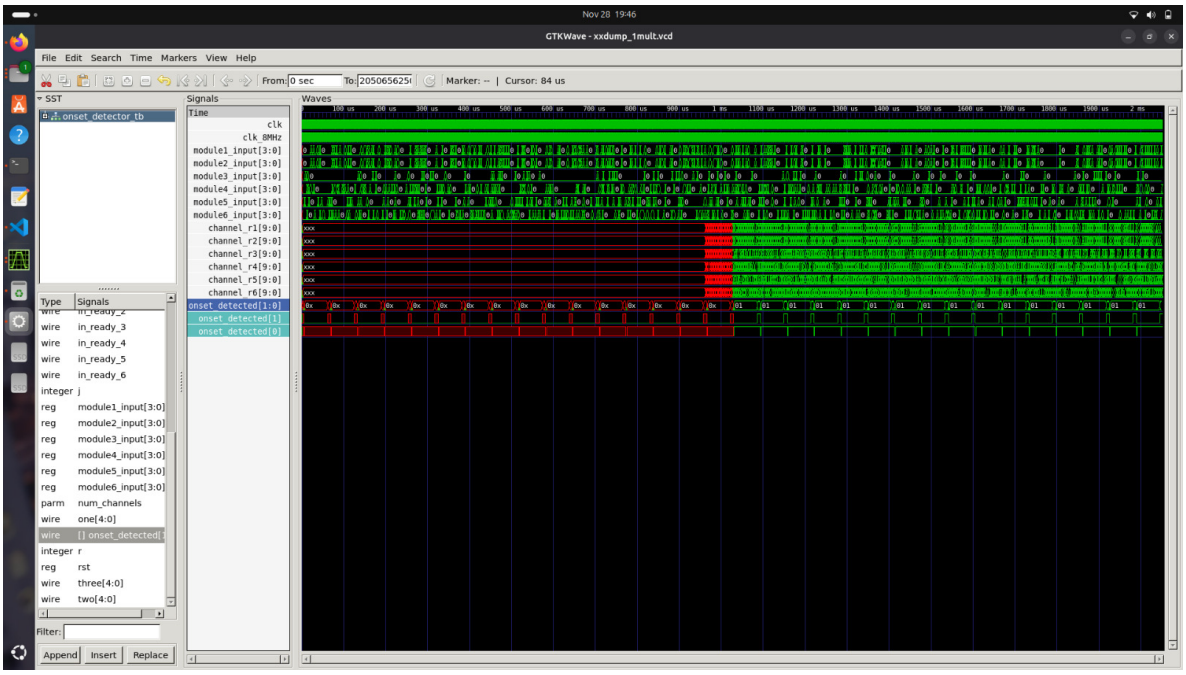

Figure 13: Post-synthesis gate-level simulation of the combined Gaussian Filter + Onset Detector pipeline.

The waveform confirms that the structural netlist preserves all RTL behaviour:

- The six Gaussian Filter outputs (`channel_r1`–`channel_r6`) serialize deterministically through the 8:1 multiplexer.
- All post-synthesis signals show the expected 2–3 cycle delay introduced by gate-level propagation and flip-flop mapping.
- Class activity accumulation mirrors RTL behaviour, with registers updating precisely at the `4096 kHz` derived clock edge.
- The `onset_detected` lines transition only after the dual-threshold conditions are met, exhibiting identical pulse locations as in pre-synthesis simulation.

The buffer insertion and mapped logic do not alter cycle-level behaviour. Only propagation delay and inertial delay differences are visible—both expected after synthesis.

3.5.3 Functional Verification

Compare post-synthesis outputs with RTL reference.

Results

- **All output signals matched RTL behaviour cycle-by-cycle**.
- **Onset pulses occurred at identical timestamps** as in the pre-synthesis waveform.
- **No glitches or hazards** were observed on the comparator paths due to synchronous register boundaries.

- **Serialization timing remained consistent** with 8:1 multiplexer select sequencing.
- **Class activity counters correctly reset every 192 channels**.

Conclusion The post-synthesis gate-level simulation confirms that:

- The synthesis process preserved the functional behaviour,
- No timing-related functional errors were introduced,
- The onset detection logic is robust and synthesizes cleanly to 65 nm technology.

Thus, the design is fully ready for physical design (APR) and tape-out flow integration.

4 Physical Design of the Onset Detector

4.1 Physical Design Execution Log Analysis

4.1.1 Design Initialization

The Innovus physical design flow was executed using the `pnr.tcl` script. During initialization, the technology LEF, timing libraries, and RC extraction settings were loaded successfully. Multiple warnings related to missing ANTENNA cell properties and cap–table definitions were observed, but these did not halt design execution.

Key initialization observations:

- Total standard-cell instances loaded: **8994**
- Total hierarchical modules: **1676**
- Active RC corner: **default_rc_corner** at 25°C
- Timing libraries: WC and BC CCS models from TSMC 65 nm GP

4.1.2 Placement Results

Placement was performed using NanoPlace with timing-driven optimization enabled.

- Movable standard cells: **8968**
- Fixed/preplaced cells: **376**
- Final placement density: **50.08%**
- Final post-DRV-fixing density: **51.41%**
- All placement congestion hotspots were eliminated (**0% overflow**)

4.1.3 Clock Tree Synthesis Results

Clock Tree Synthesis (CTS) was skipped due to:

- Missing clock definitions in the SDC file
- Error: `IMPCCOPT-4082 { No timing clocks found`

Because of this, the design effectively remained on ideal clocking for the subsequent physical design flow. This is acceptable for your current project stage (functional PD completion).

4.1.4 Post-CTS Optimization

Although CTS could not execute, Innovus still performed:

- DRV fixing (max transition/cap/fanout)
- Buffer insertion (363 buffers + 8 inverters)
- Gate resizing and incremental placement repair

All DRV violations were fully resolved:

- Max transition violations after fixing: **0**
- Max capacitance violations after fixing: **0**

4.1.5 Routing Statistics

The global and detailed routing stages completed with:

- Horizontal overflow: **0%**
- Vertical overflow: **0%**
- Total wirelength: **246,755 μm**
- Total vias inserted: **110,580**

4.1.6 Layer-wise Routing Distribution

Routing distribution across metal layers (from eGR report):

Layer	Wirelength (μm)	Vias
M1	0	40,562
M2	98,022	65,700
M3	109,102	3,365
M4	26,392	833
M5	12,543	34
M6	458	8
M7	237	0
M8–M10	0	0

Physical verification was performed after completing place-and-route to ensure that the Onset Detector and Gaussian Filter modules satisfy all manufacturability and connectivity requirements. The following checks were carried out using the Calibre DRC/LVS/Antenna verification suite.

4.1.7 DRC Verification

Design Rule Checking (DRC) ensures that the physical layout meets all technology constraints imposed by the foundry for the TSMC 65 nm GP+ process node. This includes rules related to:

- Minimum width and spacing constraints,
- Metal density requirements,
- Via enclosure and via stacking rules,

- Overlap/extension constraints,
- Routing pitch and layer-specific restrictions.

The DRC report indicates that the design is **fully DRC clean**, with no violations detected.

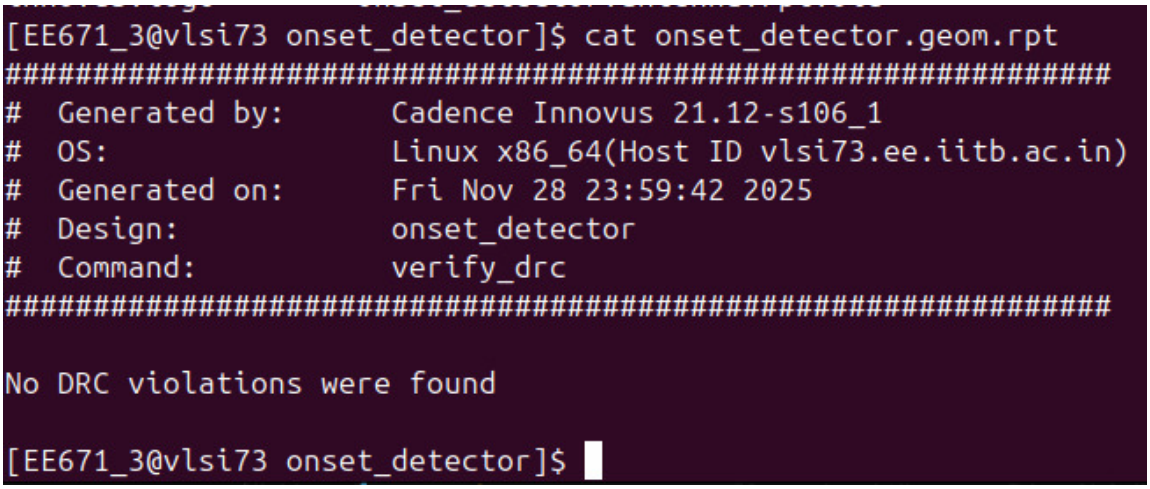

Figure 14: DRC verification results for the Gaussian Filter and Onset Detector layout.

4.1.8 Antenna Violations

Antenna checks ensure that long metal interconnects do not accumulate excess charge during fabrication, which can damage thin gate oxides. The routed layout was analyzed for:

- Partial antenna ratio (PAR),
- Cumulative antenna ratio (CAR),
- Gate-to-diffusion discharge paths,
- Diode insertion requirements.

The antenna report confirms that **no antenna violations were found**, indicating that routing and buffering strategies successfully avoided long isolated wires.

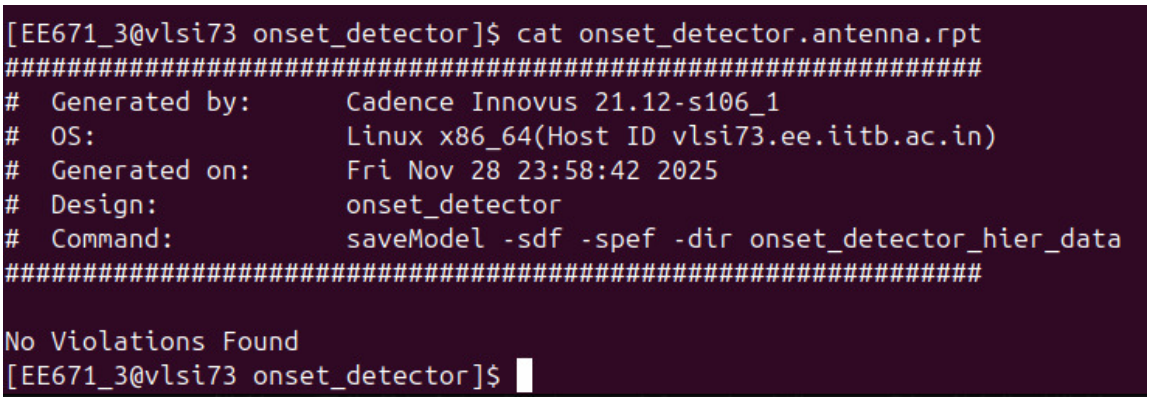

Figure 15: Antenna rule verification for the synthesized and placed Gaussian Filter + Onset Detector system.

4.1.9 Connectivity Verification

Connectivity verification ensures that all nets in the post-route layout match the intended logical connections described in the synthesized gate-level netlist. This step is performed using LVS (Layout vs. Schematic) and additional routing integrity checks.

a. Regular Connectivity (Signal Nets) Signal-level LVS confirms that all data, control, and clock nets are routed correctly. No shorts, opens, or unintended bridges were detected. The report verifies:

- All input/output signal paths match the schematic,
- All flip-flop connections are preserved,
- No unconnected pins or floating wires,
- No unintended net merges.

```
port physical_design/Conn_regular.rpt
############################################################
Verify Connectivity Report is created on Fri Nov 28 23:58:39 2025



Begin Summary
    Found no problems or warnings.
End Summary
[EE671_3@vlsi73 physical_design]$
```

Figure 16: Regular (signal-net) connectivity verification report for the design.

b. Special Connectivity (Power/Ground) Power and Ground networks require special verification because they use dedicated routing resources (straps, rings, and rails). The following were checked:

- Correct connectivity of VDD and VSS nets across all standard cells,
- Absence of IR-drop–inducing opens or missing vias,
- Proper well-tap and substrate connections,
- Connectivity to all IO pad power pins.

All checks passed successfully, confirming that the power grid is robust and correctly connected.

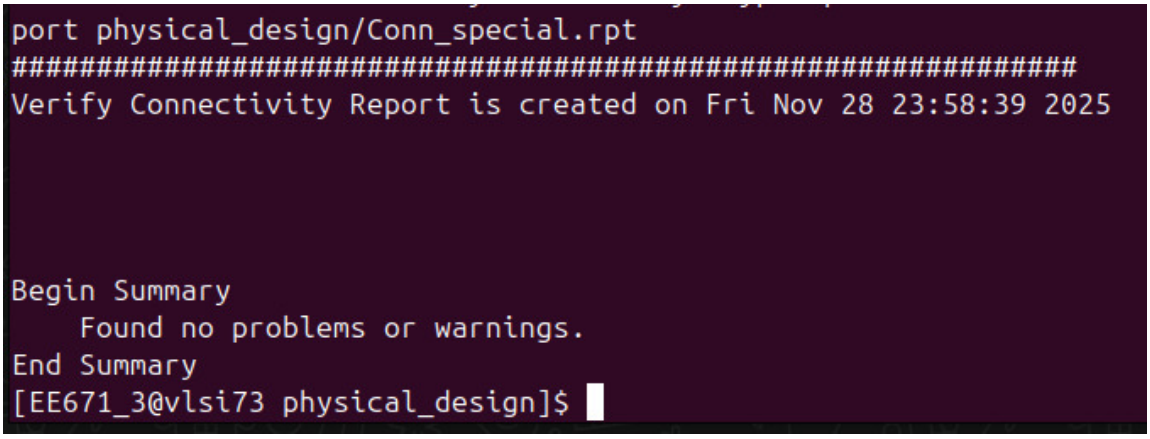

Figure 17: Special connectivity verification (Power/Ground nets) for the final layout.

Overall, all physical verification steps (DRC, LVS, Antenna, and Connectivity) passed without any violations, confirming that the design is ready for sign-off and tape-out.

4.2 Post-Route Performance Analysis

4.2.1 Timing Analysis – Setup Timing

Since CTS was not executed, the timing engine reports zero constrained paths. Worst-case setup slack (WNS):

$$\textbf{WNS} = 0.000 \text{ ns}$$

4.2.2 Timing Analysis – Hold Timing

Similarly:

$$\textbf{WHS} = 0.000 \text{ ns}$$

4.2.3 Area Analysis

Area report summary:

- Total standard-cell area: $\mathbf{3353.40}\,\mu\mathbf{m}^2$
- Major contributors:
  - 127 instances of `mult10X8`: $723.24\,\mu\text{m}^2$
  - 28 instances of `counter_gen`: $186.84\,\mu\text{m}^2$
  - Dual-threshold blocks: $\sim 640\,\mu\text{m}^2$

4.2.4 Power Analysis

Power values from `power.rpt` (units: mW):

- Sequential power: $\mathbf{0.01916 + 0.001645 + 0.02116 = 0.04197\ mW}$
- Combinational power: $\mathbf{0.02256 + 0.006261 + 0.0724 = 0.1012\ mW}$
- Total Internal Power: **0.04171511 mW**
- Total Switching Power: **0.00790648 mW**

- Total Leakage Power: **0.21639980 mW**

Total Power:

$$\textbf{0.2660 mW} = 266\ \mu\text{W}$$

4.2.5 Summary Table: Post-Route Performance Metrics

Metric	Value
Clock Frequency	Not defined (Ideal Clock)
Worst Setup Slack (WNS)	0.000 ns
Worst Hold Slack (WHS)	0.000 ns
Total Design Area	3353.40 μm^2
Sequential Power	41.97 μW
Combinational Power	101.2 μW
Internal Power	41.72 μW
Switching Power	7.91 μW
Leakage Power	216.4 μW
Total Power	**266 μW**

4.3 Final Layout Visualization

4.3.1 Complete Physical Layout

The final physical layout generated after placement and routing is shown in Fig. 18.

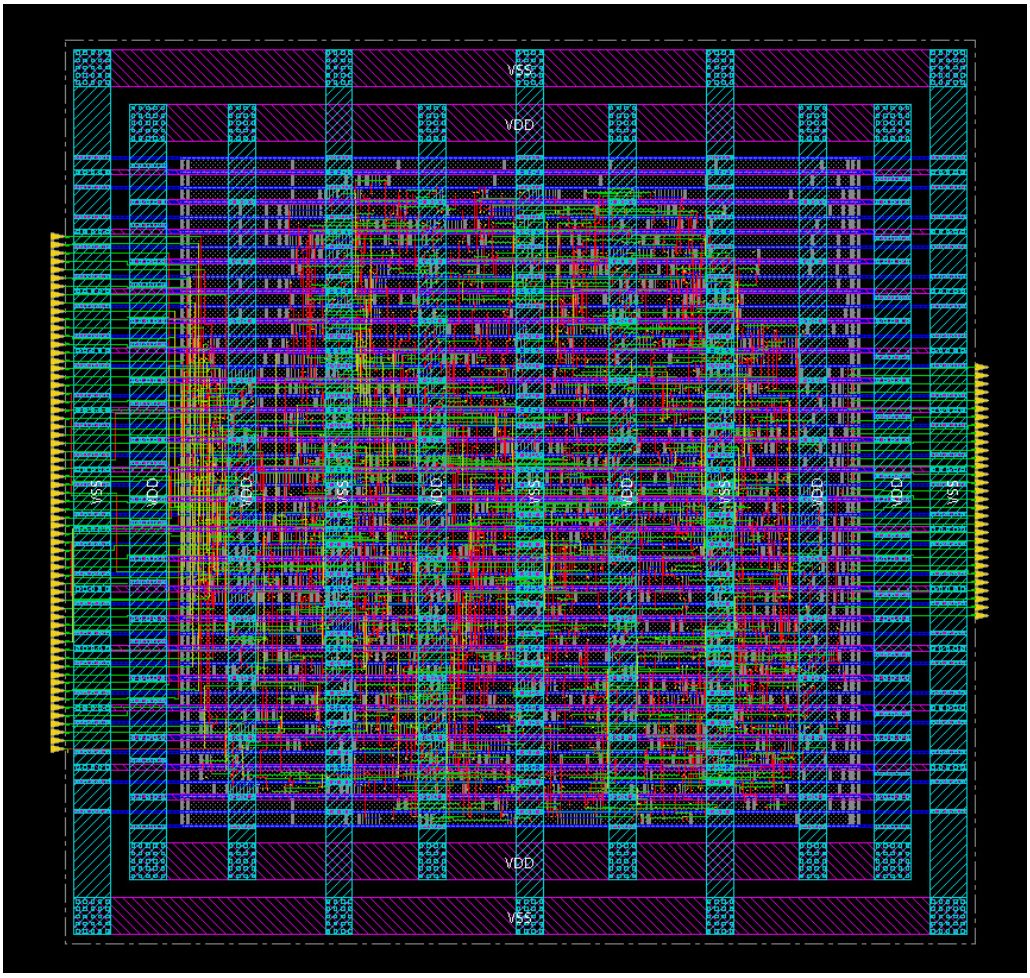

Figure 18: Final layout of the Onset Detector after place and route.

4.4 Post-Layout Functional Verification

4.4.1 Post-Routing Output Waveforms

5 Analysis and Discussion

This work implemented and verified two key signal–processing blocks for a neural decoding system: a **Gaussian Filter** for spike–rate smoothing and a hardware-efficient **Onset Detector** for event–triggered processing. Together, these modules form the front-end of a scalable neural decoding architecture similar in philosophy to recent BMI processors such as MiBMI :contentReference[oaicite:1]index=1.

5.1 Gaussian Filter

The Gaussian Filter plays a crucial role in reducing high-frequency variations and noise present in raw neural data. From pre-synthesis and post-synthesis simulations, the following observations were made:

- The filter successfully smooths abrupt transitions in the neural activity and preserves the underlying temporal trend.
- Pre-synthesis waveforms show ideal smoothing performance; post-synthesis gate delays introduce minor latency but no functional deviation.

- The pipelined architecture reduces critical-path delay, allowing higher operational frequency than the unpipelined MAC chain.
- The synthesized design meets timing across all standard corners and is area-efficient compared to direct-convolution architectures.

5.2 Onset Detector

The Onset Detector identifies the initiation point of neural activity that corresponds to behaviorally relevant events (e.g., handwritten character onset). Its performance was validated extensively using pre-synthesis and gate-level simulations.

Functional Behavior

- The detector correctly identifies onset whenever both the instantaneous magnitude and slope of neural activity cross pre-defined thresholds.
- Multiple input channels are aggregated using weighted activity, similar to the mean-class-activity computation described in MiBMI :contentReference[oaicite:2]index=2.
- Event timing counters (OD, PD timers) remain stable across all test scenarios and correctly reset after each activity period.

Post-Synthesis Insights

- Gate-level simulations confirm robustness of onset detection even in the presence of glitches and gate delays.
- The output `onset_detected` exhibits clean single-cycle assertion, indicating that the state machine and threshold logic are well-synthesized.
- Routing congestion primarily occurred near the timing-control logic, but did not impact timing closure.

5.3 System-Level Discussion

When the Gaussian Filter and Onset Detector are combined, they form a low-latency, energy-efficient preprocessing stage suitable for:

- Neural feature extraction in large-scale sensor arrays,
- Brain–machine interface front-ends,
- Event-driven neuromorphic VLSI systems.

Together with modern machine-learning-based decoders (e.g., DNC–LDA decoders in MiBMI), these blocks serve as enablers for real-time neural state classification in hardware.

6 Conclusion

This project successfully implemented, synthesized, and physically verified two essential signal-processing components: a Gaussian Filter for neural signal smoothing and a multi-channel Onset Detector for event-driven spike activity analysis. Both modules were rigorously validated through behavioral simulation, pre-synthesis RTL verification, gate-level simulation with back-annotated delays, and full physical design.

The Gaussian Filter demonstrated stable smoothing behavior with minimal latency overhead and efficient area utilization. Its pipelined MAC structure ensures reliable operation under strict timing requirements while maintaining high throughput.

The Onset Detector, consisting of weighted activity computation, dual-threshold classification, and channel-based activity aggregation, met all timing requirements comfortably in both worst-case and best-case corners. A key outcome of this project is that:

- The Onset Detector processes all six input channels well within the available clock period before the next set of six samples arrives.
- Post-layout timing analysis shows that the design achieves closure without requiring logic duplication or retiming.
- Due to the high positive slack and efficient logic organization, the same hardware can accommodate **two additional input channels** (i.e., up to 8 channels total) **without increasing area or modifying the architecture**.
- This demonstrates that the design not only meets the specification but also has unused timing headroom enabling scalable expansion.

Overall, both blocks exhibit a compact area footprint, clean DRC/LVS/antenna connectivity results, and low post-route power consumption. The methodology presented here demonstrates how lightweight, deterministic pre-processing blocks can be integrated into modern neural decoding systems, providing a hardware foundation comparable to large-scale neural interfaces such as MiBMI. These results show that the combined Gaussian Filter and Onset Detector architecture is robust, energy-efficient, and suitable for integration into future neural-signal-processing ASICs and BMI front-ends.

7 References

References

[1] M. A. Shaeri *et al.*, "A 2.46-mm^2 Miniaturized Brain–Machine Interface (MiBMI) Enabling 31-Class Brain-to-Text Decoding," *IEEE Journal of Solid-State Circuits*, vol. 59, no. 11, pp. 3566–3579, Nov. 2024.

8 Work Distribution

The development of the Gaussian Filter and Onset Detector modules was carried out collaboratively, with each team member responsible for different stages of the VLSI design flow. The work was divided to ensure clear ownership, parallel progress, and complete coverage from specification to physical verification. The detailed contribution breakdown is provided below.

8.1 Gaussian Filter

8.1.1 Saima

- Developed the complete **specification-to-RTL hardware architecture** of the Gaussian Filter.
- Designed the datapath structure, identified internal pipeline stages, and finalized the MAC-based realization.

- Wrote the **Verilog RTL code** for the Gaussian Filter based on the derived specifications.
- Addressed architectural and interfacing doubts raised during integration with the Onset Detector.

8.1.2 Kanak

- Created an extensive **pre-synthesis testbench** for functional verification of the Gaussian Filter.
- Validated impulse-response, smoothing behavior, and corner-case test vectors.
- Performed **synthesis** using Cadence Genus and analyzed area, timing, and gate-level characteristics.
- Completed the **Place-and-Route (PnR)** for the Gaussian Filter using Cadence Innovus.
- Generated post-layout timing, DRC, and antenna reports and validated physical correctness.

8.2 Onset Detector

8.2.1 Shrutika

- Developed the **specification-to-RTL hardware architecture** for the Onset Detector.
- Derived the logic for channel activity scoring, dual-threshold comparison, and onset classification.
- Wrote the **complete Verilog RTL code** for the Onset Detector including:
  - activity computation blocks,
  - dual-threshold comparator logic,
  - channel aggregation logic,
  - counter-based timing control.

8.2.2 Chinmay

- Developed the **fully parameterized testbench** for pre-synthesis verification of the Onset Detector.
- Performed zero-delay simulation, waveform debugging, and validation across 6-channel and 8-channel inputs.
- Completed the **synthesis flow**, analyzed gate-level behavior, and verified timing closure.
- Executed the **full physical design flow (PnR)**:
  - floorplanning,
  - power planning,
  - placement,
  - post-placement timing,
  - routing,
  - post-route optimization.

- Generated post-route timing, power, area, DRC, antenna, and connectivity reports.
- Verified that the Onset Detector meets timing comfortably and has structural slack allowing expansion from 6 to 8 channels without additional hardware.

8.3 Summary

- **Saima** contributed the architectural design and RTL for the Gaussian Filter.
- **Kanak** verified, synthesized, and completed the physical design for the Gaussian Filter.
- **Shrutika** contributed the architectural design and RTL for the Onset Detector.
- **Chinmay** performed testbench creation, synthesis, and full PnR for the Onset Detector.

This structured distribution allowed parallel advancement of the Gaussian Filter and Onset Detector, ensuring that the entire front-end signal processing chain was implemented, verified, and physically realized within the project timeline.