

Moving Average Filter

Smart Low-Power Proximity Sensor SoC

Comprehensive Code Dissection

Authors: Jonathan Farah, Jason Qin Github Repository: [git@github.com:jonathan-farah/Sensors_and_Security.git](https://github.com/jonathan-farah/Sensors_and_Security.git)

Statement of Purpose

Module Mission: Provide real-time digital signal processing for sensor data by implementing a configurable moving-average filter that reduces noise while maintaining hardware efficiency and low power consumption.

Why This Module Exists:

- **Problem:** Raw sensor data contains noise that causes false detections
- **Solution:** Moving average filter smooths data by averaging recent samples
- **Benefit:** Improves detection accuracy while remaining reconfigurable

Target Application: Low-power proximity sensor SoC requiring adaptive noise filtering

Design Philosophy

Core Principles:

1. **Configurability:** Runtime adjustment of filter length (1-15 taps)
2. **Efficiency:** Circular buffer eliminates data shifting overhead
3. **Predictability:** State machine provides deterministic timing
4. **Safety:** Proper initialization prevents invalid outputs
5. **Integration:** Standard handshake protocol for easy system integration

Trade-offs Made:

- Sequential accumulation (lower power) vs. parallel tree (lower latency)
 - Hardware division (flexibility) vs. shift-only (area savings)
 - Pipelined operation (throughput) vs. single-cycle (complexity)
-

Module Overview

High-Level Function: Accepts streaming sensor data, maintains a sliding window of recent samples, computes the average, and outputs filtered results.

Key Features:

- Configurable filter length (1-15 taps)

- Hardware-efficient circular buffer implementation
 - Pipelined operation with state machine control
 - Handshake-based data flow control
 - Asynchronous reset support
-

Code Dissection: Module Declaration (Lines 8-11)

```
module moving_average_filter #(
    parameter DATA_WIDTH = 32,
    parameter MAX_TAPS = 15
) (
```

Line-by-Line Analysis:

- **Line 8:** Module name establishes clear purpose
- **Line 9:** `DATA_WIDTH = 32` supports standard sensor data widths
- **Line 10:** `MAX_TAPS = 15` balances smoothing capability vs. hardware cost
- **Design Note:** Parameters allow synthesis-time customization for different applications

Hardware Impact:

- `MAX_TAPS` directly affects register count: $15 \times 32\text{-bit} = 480$ flip-flops
 - Larger `MAX_TAPS` = more smoothing but more silicon area
-

Code Dissection: Port Declarations (Lines 12-27)

Clock and Reset (Lines 13-14):

```
input  logic clock,
input  logic reset,
```

- Synchronous operation with asynchronous reset
- Reset is active-high (common in SoC designs)

Control Interface (Lines 17-18):

```
input  logic enable,
input  logic [3:0] num_taps,
```

- `enable`: Power gating support - filter can be disabled
 - `num_taps`: 4-bit value (0-15) sets active filter length **at runtime**
-

Code Dissection: Data Interface (Lines 20-27)

Input Handshake:

```
input  logic data_valid,  
input  logic [DATA_WIDTH-1:0] data_in,  
output logic data_ready,
```

- **data_valid**: Upstream asserts when new sample available
- **data_in**: The sensor sample to filter
- **data_ready**: This module signals readiness to accept data

Output Handshake:

```
output logic result_valid,  
output logic [DATA_WIDTH-1:0] result_out,  
output logic busy
```

- **result_valid**: Pulses high when filtered result ready
- **result_out**: The averaged/smoothed output
- **busy**: Indicates filter is computing (cannot accept new data)

Code Dissection: Internal Signals (Lines 32-37)

```
logic [DATA_WIDTH-1:0] delay_line [MAX_TAPS-1:0];  
logic [DATA_WIDTH-1:0] sum_accumulator;  
logic [DATA_WIDTH-1:0] filtered_result;  
logic [3:0] write_ptr;  
logic [3:0] valid_samples;  
logic computing;
```

Purpose of Each Signal:

- **delay_line**: Array storing the last MAX_TAPS samples (circular buffer)
- **sum_accumulator**: Running sum during accumulation phase
- **filtered_result**: Final averaged value (sum ÷ num_taps)
- **write_ptr**: Index where next sample will be written (0-14)
- **valid_samples**: Count of samples received (for initialization)
- **computing**: Internal flag indicating filter is processing

Code Dissection: State Machine Type (Lines 40-48)

```
typedef enum logic [1:0] {  
    IDLE          = 2'b00,
```

```

    ACCUMULATE = 2'b01,
    DIVIDE     = 2'b10,
    OUTPUT     = 2'b11
} filter_state_t;

filter_state_t current_state, next_state;
logic [3:0] accumulate_counter;

```

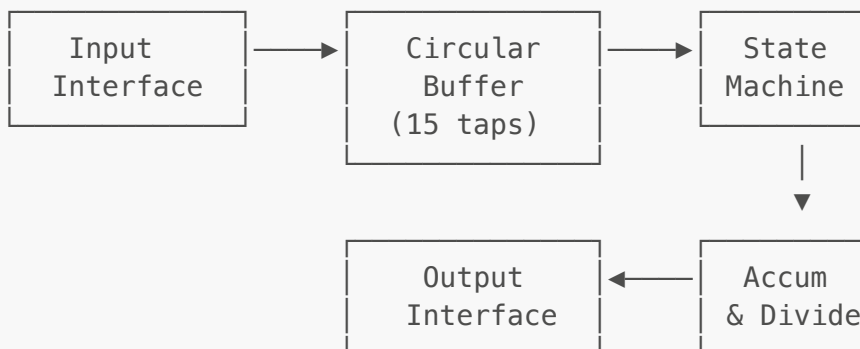
State Encoding:

- 2-bit encoding minimizes state register count
- Explicit binary values (not automatic) for simulation clarity

State Variables:

- **current_state**: Registered current state
- **next_state**: Combinational next state (computed each cycle)
- **accumulate_counter**: Tracks progress through buffer (0 to num_taps)

Core Architecture



Code Dissection: Delay Line Block (Lines 53-76) - Part 1

```

always_ff @(posedge clock or posedge reset) begin
    if (reset) begin
        for (int i = 0; i < MAX_TAPS; i++) begin
            delay_line[i] <= '0;
        end
        write_ptr <= 4'h0;
        valid_samples <= 4'h0;
    end
end

```

Reset Behavior (Lines 54-59):

- **Line 54**: Asynchronous reset - highest priority
- **Lines 55-57**: Clear all 15 buffer entries to zero

- Uses **for** loop (synthesizes to parallel resets)
- **'0** notation: all bits zero regardless of width
- **Line 58:** Reset write pointer to position 0
- **Line 59:** Clear valid sample counter

Why Asynchronous Reset?: Power-on initialization without clock

Code Dissection: Delay Line Block (Lines 53-76) - Part 2

```
end else if (enable && data_valid && !computing) begin
    // Write new sample to circular buffer
    delay_line[write_ptr] <= data_in;

    // Update write pointer (circular)
    if (write_ptr == 4'hF) begin
        write_ptr <= 4'h0;
    end else begin
        write_ptr <= write_ptr + 4'h1;
    end
end
```

Sample Writing (Lines 60-69):

- **Line 60:** Three conditions for accepting data:
 1. **enable** - module is enabled
 2. **data_valid** - upstream has new data
 3. **!computing** - filter is not busy processing
 - **Line 62:** Store sample at current write position
 - **Lines 65-69:** Circular pointer increment
 - If at position 15 (0xF), wrap to 0
 - Otherwise increment by 1
 - **Critical:** Enables continuous operation without buffer shifts
-

Code Dissection: Delay Line Block (Lines 53-76) - Part 3

```
    // Track valid samples
    if (valid_samples < num_taps) begin
        valid_samples <= valid_samples + 4'h1;
    end
end
end
```

Initialization Counter (Lines 72-74):

- **Purpose:** Prevents filtering until buffer has enough samples
- **Line 72:** Only increment if below target tap count

- **Line 73:** Increment valid sample count
- **Behavior:**
 - Counts from 0 to num_taps
 - Stops incrementing once buffer is "full"
 - Example: If num_taps=8, stops at 8

Design Insight: This prevents outputting invalid results during cold start

Circular Buffer Implementation Summary

Why Circular Buffer?

- **Alternative 1:** Shift all data every cycle → wasteful
- **Alternative 2:** FIFO with read/write pointers → more complex
- **Chosen:** Circular buffer with single write pointer

Benefits:

- Constant-time operations ($O(1)$)
- No data movement required
- Minimal control logic
- Easy to read any sample position

Hardware Cost:

- 15 × 32-bit registers = 480 flip-flops
 - 4-bit write pointer = 4 flip-flops
 - 4-bit sample counter = 4 flip-flops
 - **Total:** 488 flip-flops for buffer management
-

Code Dissection: State Register (Lines 81-87)

```
always_ff @(posedge clock or posedge reset) begin
    if (reset) begin
        current_state <= IDLE;
    end else begin
        current_state <= next_state;
    end
end
```

State Register Logic:

- **Line 81:** Synchronous state update on clock edge
- **Line 82-83:** Reset forces state machine to IDLE
- **Line 85:** Normal operation: register next_state
- **Pattern:** Classic two-process state machine
 - This process: sequential (registered)

- Next process: combinational (next-state logic)

Why Split?: Prevents combinational loops and clarifies timing

Code Dissection: Next State Logic (Lines 89-115) - Part 1

```
always_comb begin
    next_state = current_state; // Default: stay in current state

    case (current_state)
        IDLE: begin
            if (enable && data_valid && valid_samples == num_taps) begin
                next_state = ACCUMULATE;
            end
        end
    end
```

IDLE State (Lines 93-97):

- **Line 90:** Default assignment prevents latches
 - **Line 92:** Case statement based on current state
 - **Line 94:** Transition conditions:
 1. **enable**: Module enabled
 2. **data_valid**: New data available
 3. **valid_samples == num_taps**: Buffer is full
 - **Line 95:** Move to ACCUMULATE when all conditions met
 - **Implicit:** Stay in IDLE if conditions not met
-

Code Dissection: Next State Logic (Lines 89-115) - Part 2

```
        ACCUMULATE: begin
            if (accumulate_counter >= num_taps) begin
                next_state = DIVIDE;
            end
        end

        DIVIDE: begin
            next_state = OUTPUT;
        end

        OUTPUT: begin
            next_state = IDLE;
        end
```

ACCUMULATE State (Lines 99-103):

- Waits until all samples are summed

- When counter reaches num_taps → go to DIVIDE

DIVIDE State (Lines 105-107):

- Always transitions to OUTPUT after one cycle
- Single-cycle division (or shift)

OUTPUT State (Lines 109-111):

- Always returns to IDLE after one cycle
- Ready for next filtering operation

State Machine Design Summary

Four-State FSM:

1. **IDLE** - Wait for valid input data and full buffer
2. **ACCUMULATE** - Sum all tap values sequentially
3. **DIVIDE** - Calculate average (sum ÷ num_taps)
4. **OUTPUT** - Present result for one cycle

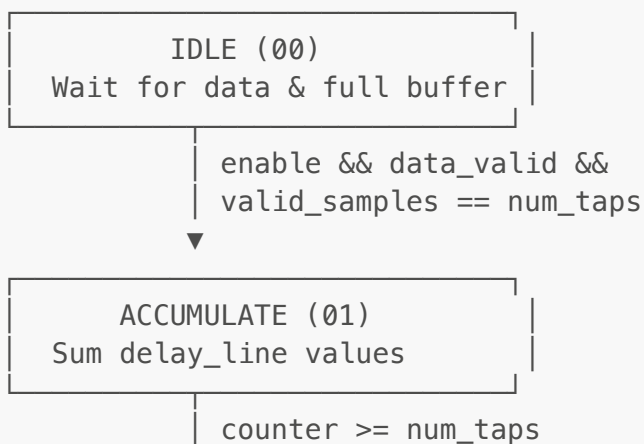
State Transition Flow:

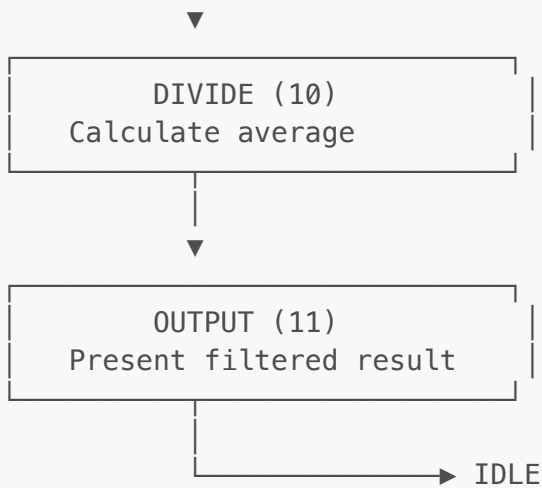
IDLE → ACCUMULATE → DIVIDE → OUTPUT → IDLE (repeat)

Timing Analysis:

- IDLE: Variable (waits for data)
- ACCUMULATE: num_taps cycles
- DIVIDE: 1 cycle
- OUTPUT: 1 cycle
- **Total per result:** num_taps + 2 cycles

State Transitions





Code Dissection: Accumulation Logic (Lines 120-170) - Part 1

```

always_ff @(posedge clock or posedge reset) begin
  if (reset) begin
    sum_accumulator <= '0;
    accumulate_counter <= 4'h0;
    filtered_result <= '0;
  end else begin
    case (current_state)

```

Reset Behavior (Lines 121-125):

- **Lines 122-124:** Initialize all computation registers to zero
 - **sum_accumulator:** Clears running sum
 - **accumulate_counter:** Resets loop counter
 - **filtered_result:** Clears output register
- **Line 126:** State-based behavior using case statement

Purpose: Registered outputs prevent glitches and ensure timing closure

Code Dissection: IDLE State Behavior (Lines 127-135)

```

begin
  IDLE: begin
    sum_accumulator <= '0;
    accumulate_counter <= 4'h0;
    if (enable && data_valid && valid_samples == num_taps)

      // Start accumulation on next cycle
      sum_accumulator <= delay_line[0];
      accumulate_counter <= 4'h1;
    end
  end
end

```

IDLE Preparation (Lines 127-135):

- **Lines 128-129:** Default: clear accumulator and counter
- **Line 130:** Check if ready to start filtering
- **Line 132:** Pre-load first sample into accumulator
- **Line 133:** Set counter to 1 (sample 0 already loaded)

Design Note: This eliminates one cycle from accumulation loop

- Instead of $0 + \text{sample}[0] + \text{sample}[1] + \dots$, we start with $\text{sample}[0]$ loaded

Code Dissection: ACCUMULATE State Behavior (Lines 137-142)

```

    ACCUMULATE: begin
        if (accumulate_counter < num_taps) begin
            sum_accumulator <= sum_accumulator +
delay_line[accumulate_counter];
            accumulate_counter <= accumulate_counter + 4'h1;
        end
    end

```

Sequential Addition (Lines 137-142):

- **Line 138:** Guard condition - only add if more samples remain
- **Line 139: Critical operation:** Add current buffer entry to sum
 - Uses `accumulate_counter` as read index
 - Reads from circular buffer (any position)
- **Line 140:** Increment counter for next sample

Why Sequential?

- One adder instead of tree of adders
- Lower power consumption
- Smaller area
- Trade-off: More cycles but simpler hardware

Code Dissection: Accumulate Example

Example: `num_taps = 4`

Cycle	State	Counter	Operation	Accumulator Value
1	IDLE→ACC	0→1	Load <code>delay_line[0]</code>	<code>sample[0]</code>
2	ACCUMULATE	1→2	Add <code>delay_line[1]</code>	<code>sample[0]+sample[1]</code>
3	ACCUMULATE	2→3	Add <code>delay_line[2]</code>	<code>sum + sample[2]</code>

Cycle	State	Counter	Operation	Accumulator Value
4	ACCUMULATE	3→4	Add delay_line[3]	sum + sample[3]
5	DIVIDE	4	counter >= num_taps	Divide by 4

Total Accumulation Time: num_taps cycles (pre-load optimization)

Hardware: Single 32-bit adder + 32-bit register

Code Dissection: DIVIDE State Behavior (Lines 144-158)

```
DIVIDE: begin
  case (num_taps)
    4'd1: filtered_result <= sum_accumulator;
    4'd2: filtered_result <= sum_accumulator >> 1;
    4'd4: filtered_result <= sum_accumulator >> 2;
    4'd8: filtered_result <= sum_accumulator >> 3;
    default: begin
      filtered_result <= sum_accumulator / num_taps;
    end
  endcase
end
```

Division Optimization (Lines 144-158):

- **Line 145:** Switch on number of taps
- **Line 148:** 1 tap = no averaging, pass through
- **Line 149:** 2 taps = divide by 2 = right shift 1 bit
- **Line 150:** 4 taps = divide by 4 = right shift 2 bits
- **Line 151:** 8 taps = divide by 8 = right shift 3 bits
- **Lines 152-155:** All other tap counts use division operator

Why This Matters: Right shift is FREE in hardware (just wiring)

Code Dissection: Division Hardware Analysis

Power-of-2 Cases (1, 2, 4, 8):

- **Hardware:** Just wire routing, no logic gates
- **Example:** >> 2 means connect bit[31:2] to output bit[29:0]
- **Area:** Zero additional logic
- **Power:** Zero dynamic power
- **Timing:** Combinational, no delay

Non-Power-of-2 Cases (3, 5, 6, 7, 9-15):

- **Hardware:** Division requires iterative or lookup logic

- **Area:** Significant (divider can be 50-100+ gates)
- **Power:** Higher dynamic power
- **Timing:** May need multiple cycles or long critical path

Design Trade-off:

- Flexibility (support any num_taps) vs. Efficiency (shift-only)
- This design chooses flexibility with optimization for common cases

Code Dissection: OUTPUT and Default States (Lines 160-168)

```
        OUTPUT: begin
            // Hold result for one cycle
        end

        default: begin
            sum_accumulator <= '0;
            accumulate_counter <= 4'h0;
        end
    endcase
end
end
```

OUTPUT State (Lines 160-162):

- No operations - just holds filtered_result stable
- Gives downstream logic one cycle to capture result
- Result captured by output control block (next section)

Default State (Lines 164-167):

- Safety net for undefined states
- Resets computation registers
- Should never execute in normal operation
- **Best Practice:** Always include default in case statements

Code Dissection: Output Control Block (Lines 175-187)

```
always_ff @(posedge clock or posedge reset) begin
    if (reset) begin
        result_valid <= 1'b0;
        result_out <= '0;
    end else begin
        result_valid <= 1'b0;

        if (current_state == OUTPUT) begin
            result_valid <= 1'b1;
        end
    end
end
```

```

        result_out <= filtered_result;
    end
end
end

```

Output Logic (Lines 175-187):

- **Lines 177-178:** Reset clears valid flag and output data
- **Line 180: Default:** result_valid is LOW (one-cycle pulse)
- **Lines 182-185:** When in OUTPUT state:
 - Assert result_valid HIGH for one cycle
 - Transfer filtered_result to output port

Pulse Behavior: result_valid is HIGH only during OUTPUT state

- Acts as a "data ready" strobe
- Downstream can capture data when valid pulse seen

Code Dissection: Status Signals (Lines 192-194)

```

assign computing = (current_state != IDLE);
assign busy = computing;
assign data_ready = !computing && enable;

```

Status Signal Generation (Lines 192-194):

- **Line 192:** `computing` is HIGH in any non-IDLE state
 - True during ACCUMULATE, DIVIDE, OUTPUT
 - Prevents new data from being accepted during processing
- **Line 193:** `busy` directly copies `computing`
 - External status indicator
 - Lets upstream know filter is working
- **Line 194:** `data_ready` asserts when:
 - NOT computing (filter is idle), AND
 - Module is enabled
 - Tells upstream "send me data now"

Design Pattern: Ready-valid handshake protocol

Handshake Protocol Summary

Input Handshake:

Upstream		Filter
-----		-----
data_valid = 1	→	(sees valid)
data_in = X	→	(samples data)
	←	data_ready = 1 (was ready)

Output Handshake:

Filter		Downstream
-----		-----
result_valid = 1	→	(capture data)
result_out = Y	→	(read value)

Backpressure:

- If **computing = 1**, filter ignores new **data_valid**
- Upstream must wait for **data_ready** before sending
- Prevents data loss and buffer overflow

Complete Operation Example: 4-Tap Filter

Scenario: num_taps = 4, incoming samples: 10, 20, 30, 40, 50

Cycle	Event	write_ptr	valid_samples	Buffer State	State	Action
1	Sample 10	0→1	0→1	[10, -, -, -]	IDLE	Store, increment
2	Sample 20	1→2	1→2	[10,20, -, -]	IDLE	Store, increment
3	Sample 30	2→3	2→3	[10,20,30,-]	IDLE	Store, increment
4	Sample 40	3→4	3→4	[10,20,30,40]	IDLE	Store, buffer full!
5	Sample 50	4→5	4	[10,20,30,40,50]	IDLE→ACC	Trigger filter
6	-	5	4	[...]	ACCUMULATE	sum = 10
7	-	5	4	[...]	ACCUMULATE	sum = 10+20 = 30
8	-	5	4	[...]	ACCUMULATE	sum = 30+30 = 60

Cycle	Event	write_ptr	valid_samples	Buffer State	State	Action
9	-	5	4	[...]	ACCUMULATE	sum = 60+40 = 100
10	-	5	4	[...]	DIVIDE	result = 100>>2 = 25
11	Output 25	5	4	[...]	OUTPUT	result_valid=1
12	-	5	4	[...]	IDLE	Ready for next

Result: Average of [10,20,30,40] = 25 ✓

Initialization Behavior Deep Dive

Cold Start Problem:

- What if buffer has only 2 samples but num_taps = 8?
- Averaging partial data would give incorrect results

Solution - valid_samples Counter:

```
if (valid_samples < num_taps) begin
    valid_samples <= valid_samples + 1;
end
```

Process:

1. Counter starts at 0
2. Increments with each new sample (up to num_taps)
3. Filtering only starts when `valid_samples == num_taps`
4. First output is guaranteed valid

Behavior on num_taps Change:

- If user changes num_taps to smaller value → filter immediately
- If user changes num_taps to larger value → wait for more samples

Code Structure Summary

Complete File Map (197 lines total):

Lines	Section	Purpose
1-7	Header	Module description and metadata

Lines	Section	Purpose
8-27	Module Declaration	Parameters and port definitions
32-37	Internal Signals	Delay line, accumulators, counters
40-48	State Machine Type	FSM enumeration and state variables
53-76	Delay Line Management	Circular buffer with write pointer
81-87	State Register	Sequential state machine logic
89-115	Next State Logic	Combinational state transitions
120-170	Accumulation Logic	Core filtering computation
175-187	Output Control	Result valid and output register
192-194	Status Signals	Busy and ready signal generation
196	End Module	Module termination

Key Design Features Summary

✓ **Configurable:** Runtime tap adjustment (1-15) ✓ **Efficient:** Circular buffer eliminates shifting ✓ **Pipelined:** Predictable multi-cycle operation ✓ **Safe:** Proper initialization prevents invalid outputs ✓ **Low Power:** Sequential accumulation (1 adder vs. tree) ✓ **Optimized:** Power-of-2 division uses shifts (zero cost) ✓ **Robust:** Asynchronous reset, handshake protocol ✓ **Maintainable:** Clear state machine, well-commented code

Performance Characteristics

Latency:

- Buffer fill: num_taps cycles
- Processing: $\text{num_taps} + 2$ cycles
- Total first output: $2 \times \text{num_taps} + 2$ cycles

Throughput:

- New result every $\text{num_taps} + 3$ cycles
- Can be optimized with overlapping operations

Hardware Resources

Storage:

- Delay line: $\text{MAX_TAPS} \times \text{DATA_WIDTH}$ flip-flops
- Accumulator: DATA_WIDTH flip-flops
- Control: ~20 flip-flops

Logic:

- 1 adder (DATA_WIDTH bits)
- 1 divider (or shifters for power-of-2)
- State machine logic (minimal)

Use Cases

Sensor Data Filtering:

- Proximity sensor noise reduction
- Temperature sensor smoothing
- ADC output conditioning

System Integration:

- Part of larger sensor SoC
- Connects to ADC interface
- Feeds threshold detection logic

Configuration Examples

Taps	Smoothing	Latency	Use Case
1	None	Low	Pass-through / Debug
2	Minimal	Low	Fast response
4	Moderate	Medium	Balanced
8	Strong	Higher	Heavy filtering
15	Maximum	Highest	Max noise rejection

Integration Considerations

Clock Domain: Single synchronous clock **Reset:** Asynchronous reset, synchronous release **Enable:** Module-level power gating support **Error Handling:** None required (deterministic)

Timing:

- All operations registered
- No combinatorial paths input→output

Future Enhancements

Potential Improvements:

- Overlapped processing (higher throughput)
- Weighted averages (FIR filter)
- Cascadable configuration

- DMA interface for burst operation
- Statistical output (min/max/variance)

Summary: What We Dissected

Complete Code Analysis - All 197 Lines:

1. ✓ **Module Declaration** (Lines 8-27): Parameters, ports, handshake interface
2. ✓ **Internal Signals** (Lines 32-48): Buffers, state machine, counters
3. ✓ **Circular Buffer** (Lines 53-76): Sample storage with write pointer
4. ✓ **State Machine** (Lines 81-115): 4-state FSM with transitions
5. ✓ **Accumulation** (Lines 120-170): Sequential summation and division
6. ✓ **Output Control** (Lines 175-187): Result valid pulse generation
7. ✓ **Status Signals** (Lines 192-194): Ready, busy, computing flags

Key Implementation Decisions:

- Circular buffer over shifting → efficiency
- Sequential accumulation over parallel → low power
- Power-of-2 optimization → zero-cost common cases
- Valid sample counter → safe initialization

Design Lessons Learned

From This Implementation:

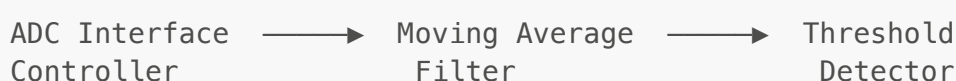
1. **Trade-offs Matter:** Sequential (low power) vs. Parallel (low latency)
2. **Optimize Common Cases:** Power-of-2 shifts save significant area
3. **Safety First:** Initialization logic prevents invalid outputs
4. **Interface Cleanly:** Standard handshakes enable easy integration
5. **Document Well:** Comments explain "why" not just "what"
6. **State Machines:** Clarify complex control flow
7. **Asynchronous Reset:** Essential for power-on initialization

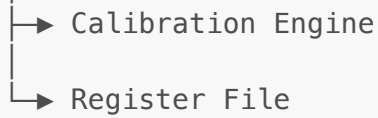
Reusable Patterns:

- Circular buffer technique
- Two-process state machine
- Ready-valid handshake
- One-hot optimization (power-of-2)

Application Context

This Filter in the SoC:



**Integration Points:**

- Receives sensor data from ADC interface controller
- Outputs to threshold detection and calibration
- Configured via register file (num_taps setting)
- Part of signal processing pipeline

Why 15 Taps?: Balances smoothing capability (max) vs. silicon area

Performance Summary

Latency:

- Buffer fill: num_taps cycles (cold start only)
- Processing: $\text{num_taps} + 2$ cycles per result
- Total first output: $2 \times \text{num_taps} + 2$ cycles
- Subsequent outputs: $\text{num_taps} + 3$ cycles

Throughput:

- Maximum: 1 result every $\text{num_taps} + 3$ cycles
- With num_taps=8: 1 result / 11 cycles
- Can be improved with overlapping (future enhancement)

Resource Usage:

- Flip-flops: ~500 (mostly buffer storage)
 - Adder: 1×32 -bit
 - Divider: 1 (if non-power-of-2 taps used)
 - State machine: Minimal logic
-

Summary: Moving Average Filter

What It Does:

- Smooths noisy sensor data by averaging recent samples
- Configurable filter length (1-15 taps) set at runtime
- Produces filtered output with predictable latency

How It Works:

- Stores samples in circular buffer (no shifting)
- State machine sequences: accumulate \rightarrow divide \rightarrow output
- Sequential accumulation minimizes power and area

- Power-of-2 optimization eliminates division logic

Why It Matters:

- Improves sensor detection accuracy
- Low power consumption for battery-operated devices
- Flexible configuration adapts to noise conditions
- Clean interface integrates easily with SoC

Perfect for: Resource-constrained sensor applications requiring adaptive filtering

Questions?

Topics Covered:

- Complete line-by-line code dissection
- Design decisions and trade-offs
- Hardware implementation details
- Integration and system context

Contact: Cognichip Co-Designer Team

Documentation: See module comments in [moving_average_filter.sv](#)

Repository: Sensors_and_Security project

Thank You

Module: [moving_average_filter.sv](#) (197 lines dissected) **Project:** Smart Low-Power Proximity Sensor SoC **Authors:** Jonathan Farah, Jason Qin

Understanding every line of RTL creates better hardware engineers