Adaptive Noise Cancellation

Sharvani Vadlamani (sv2734), Sayem Kamal (sk5336), Huda Jafri (shj2127)

May 2025

Contents

1 Introduction **3**

2 System Block Diagram **3**

3 Algorithm: LMS Adaptive Filtering **4**
- 3.1 Adaptive Noise Cancellation Theory 4
- 3.2 Causality and Convergence Conditions 4
- 3.3 Fixed-Point Number Representation 4

4 Hardware Design **5**
- 4.1 Audio Clock and I^{2}S Interface 5
- 4.2 LMS Datapath 5
- 4.3 Avalon-MM Register Interface 5
- 4.4 Platform Designer Integration 6

5 Software Design **6**
- 5.1 Linux Kernel Module 6
- 5.2 Codec Initialization (`nc_codec.c`) 7
- 5.3 Userspace Application (`nc_demo.c`) 7
- 5.4 Monitor (`nc_monitor.c`) 7

6 Hardware/Software Interface **8**

7 Verification and Testing **8**
- 7.1 Algorithm Correctness in Simulation 8
- 7.2 I^{2}S Framing Correctness 8
- 7.3 Timing and Synthesis 8
- 7.4 Peripheral Reachability and Codec Bring-Up 9
- 7.5 Audio Path Verification 9
- 7.6 Noise Cancellation Performance 9

8 Failure Modes **10**
- 8.1 PLL Not Locked 10
- 8.2 LMS Divergence 10
- 8.3 Insufficient Causal Delay Between Channels 10
- 8.4 I^{2}S Format Mismatch 10
- 8.5 LRCK Polarity Flip 10

9 Results **10**

10 Lessons Learned **11**

11 References 11

12 Code 11

12.1 FPGA Audio Peripheral . . . 11
12.1.1 noise_cancel_top.sv . . . 11
12.1.2 lms_filter.sv . . . 14
12.1.3 avalon_slave.sv . . . 20
12.1.4 i2s_adc_rx.sv . . . 22
12.1.5 i2s_dac_tx.sv . . . 24
12.1.6 i2c_controller.sv . . . 25
12.2 Linux Kernel Interface . . . 27
12.2.1 noise_cancel.h . . . 27
12.3 HPS Userspace Software . . . 29
12.3.1 nc_codec.c . . . 29
12.3.2 nc_demo.c . . . 30
12.3.3 nc_monitor.c . . . 35
12.4 Verification . . . 37
12.4.1 tb_lms.sv . . . 37
12.4.2 tb_lms.cpp . . . 37
12.4.3 tb_i2s.sv . . . 42
12.4.4 tb_i2s.cpp . . . 43

1 Introduction

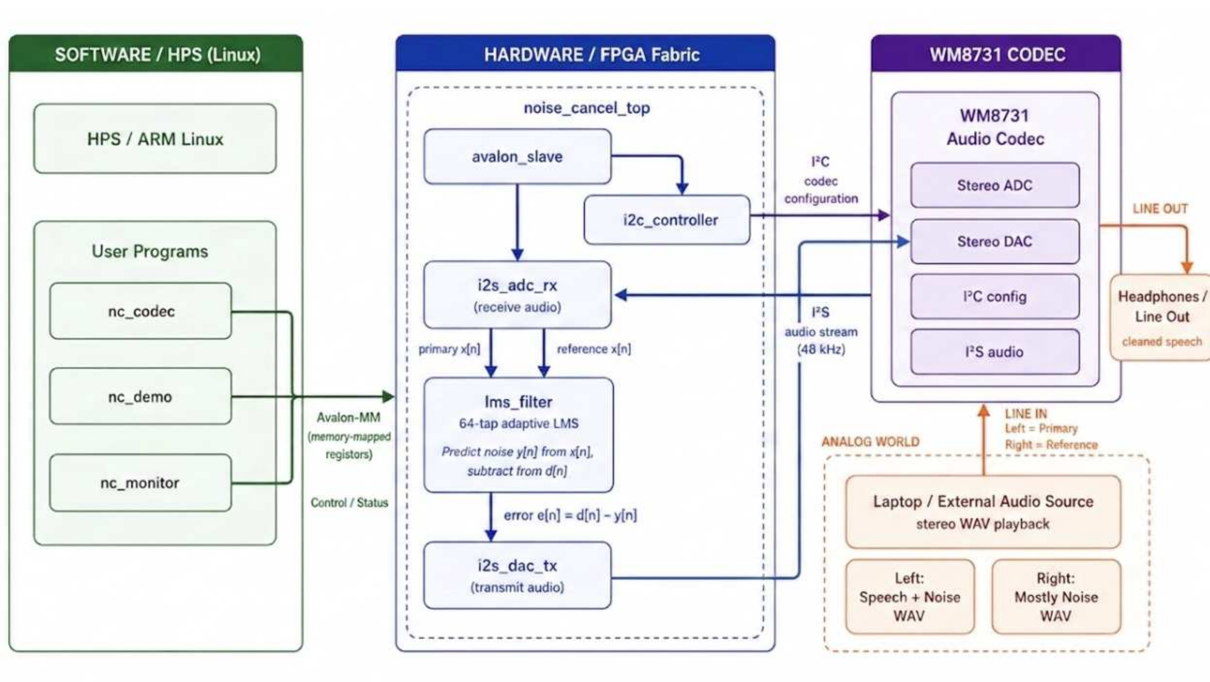
For this project, we built a real-time adaptive noise cancellation (ANC) system on the Terasic DE1-SoC FPGA board. The system reduces background noise from a primary audio channel by exploiting a separate reference channel that contains mostly noise. We recorded two synchronized WAV files in the same room, one close-up track capturing speech mixed with background noise (primary), and one distant track capturing mostly the ambient noise (reference). These were combined into a single stereo file and played back from a laptop into the board's LINE IN jack via a standard 3.5 mm TRS cable.

This approach is functionally identical to a live two-microphone setup — the WM8731 codec and FPGA see the same stereo L/R signal structure — while offering the advantages of the input being perfectly repeatable for demos and testing, and channel levels being easily verified during post-production.

Both channels enter the board through the on-board WM8731 audio CODEC. The FPGA runs a 64-tap Least Mean Squares (LMS) adaptive filter at 48 kHz, continuously estimating the noise from the reference channel and subtracting that estimate from the primary signal. The cleaned output is routed through LINE OUT to headphones.

Control and monitoring are handled by a Linux userspace application running on the ARM HPS (Hard Processor System), communicating with the FPGA fabric through a custom Avalon-MM peripheral. A kernel module exposes the peripheral as `/dev/noise_cancel`, providing ioctls for setting the LMS step size, toggling bypass mode, and reading back learned filter coefficients and live sample snapshots.

2 System Block Diagram



The system has three major subsystems: (1) the FPGA audio and LMS datapath running at 12.288 MHz, (2) the Avalon-MM register interface bridging FPGA and HPS, and (3) the Linux kernel module and userspace application running on the ARM cores.

3 Algorithm: LMS Adaptive Filtering

3.1 Adaptive Noise Cancellation Theory

The Least Mean Squares (LMS) algorithm is the core of the noise cancellation. Given:

- $d[n]$ — the *primary* signal (speech + noise), played on the LEFT stereo channel
- $x[n]$ — the *reference* signal (mostly noise), played on the RIGHT stereo channel
- $\mathbf{w}[n]$ — the adaptive filter weight vector of length $N = 64$

The filter output (noise estimate) is:

$$y[n] = \sum_{k=0}^{N-1} w_k[n] \cdot x[n-k]$$

The error (cleaned output) is:

$$e[n] = d[n] - y[n]$$

The weight update rule (steepest-descent on the mean square error) is:

$$w_k[n+1] = w_k[n] + \mu \cdot e[n] \cdot x[n-k], \quad k = 0, \ldots, N-1$$

where μ is the adaptation step size. As the algorithm converges, $y[n]$ becomes a close estimate of the noise component in $d[n]$, so $e[n]$ converges toward the speech-only signal.

3.2 Causality and Convergence Conditions

For the LMS to converge, the reference channel must observe each noise event *no later than* the primary channel. Our 64-tap filter at 48 kHz spans $64/48000 \approx 1.33$ ms, which corresponds to roughly 45 cm of sound travel at room temperature. When recording, the distant (reference) microphone was positioned closer to the noise source than the primary microphone, with a path-length differential of ≤ 30 cm, satisfying the causality requirement.

The LMS stability condition on μ is:

$$0 < \mu < \frac{2}{N \cdot E[x^2]}$$

For full-scale Q1.15 input, this gives an upper bound of approximately $2/64 \approx 0.03$. Our default $\mu =$ `0x0200` in Q0.16 (≈ 0.0078) is well below this bound, preventing divergence on startup.

3.3 Fixed-Point Number Representation

All arithmetic is performed in fixed-point to fit within FPGA DSP blocks:

Signal	**Format**	**Bits**	**Notes**
$d[n]$, $x[n]$, $e[n]$	Q1.15	16	Signed, range ± 1
Weights w_k	Q2.30	32	Signed, range ± 2
MAC accumulator	Q2.46	64	$Q1.15 \times Q2.30 = Q3.45$, sign-extended
μ	Q0.16	16	Unsigned
$\mu \cdot e[n]$	Q1.31	32	Intermediate product

All narrowing operations (e.g. the MAC accumulator back to Q1.15) use saturating arithmetic and set a sticky `STATUS.saturation` flag if overflow occurs.

4 Hardware Design

4.1 Audio Clock and I^{2}S Interface

The WM8731 audio codec requires a master clock (`AUD_XCK`) of 12.288 MHz. We instantiate an Altera PLL IP block (`audio_pll_0`) in Platform Designer that takes the 50 MHz board oscillator and produces this frequency. The clock hierarchy is:

Clock	Rate	Source	Purpose
`clk_50`	50 MHz	Board oscillator	PLL reference
`aud_mclk`	12.288 MHz	`audio_pll_0`	Codec master clock; all FPGA audio logic
`aud_bclk`	3.072 MHz	`aud_mclk / 4`	I^{2}S bit clock
`aud_lrck`	48 kHz	`aud_bclk / 64`	I^{2}S word/frame clock

The I^{2}S receiver (`i2s_adc_rx`) deserializes the codec's ADC output into 16-bit left and right samples. On each `sample_valid` pulse (once per LRCK period, i.e. 48 kHz), the LMS datapath receives a new $\langle d[n], x[n] \rangle$ pair. The I^{2}S transmitter (`i2s_dac_tx`) reads from a one-deep output register that the LMS loads at the end of each update sweep, introducing a fixed one-frame latency ($\approx$ 20.83 μs) that is acoustically inaudible.

The codec is configured over I^{2}C through an FPGA-side I^{2}C controller clocked by `aud_mclk`. The kernel module's `NC_I2C_WRITE` ioctl drives this controller from the HPS. Because the WM8731 control interface is write-only (no read transactions defined in the datasheet), the kernel module maintains a 16-entry shadow table in debugfs that reflects the last-written value of each codec register.

4.2 LMS Datapath

The LMS filter core operates on the `aud_mclk` clock (12.288 MHz), giving 256 clock cycles per sample period at 48 kHz. The datapath shares a single pipelined multiplier between the MAC sweep and the weight-update sweep to minimize DSP block usage.

Operation	Cycles	Notes
Latch $d[n]$, $x[n]$ from I^{2}S RX	1	On `sample_valid`
Shift-register update + MAC sweep $y = \sum w_k x[n-k]$	66	1 prologue + 64 taps + 1 epilogue
Compute $e[n] = d[n] - y$, saturating	1	
Pre-compute $\mu_e = \mu \cdot e[n]$	1	Splits the 3-operand product
Weight update sweep $w_k \mathrel{+}= \mu_e \cdot x[n-k]$	66	Reuses same multiplier
Push $e[n]$ to output register	1	
Total	$\approx$ **136**	of 256 available ($\approx$47% slack)

The sample shift register (storing $x[n], x[n-1], \ldots, x[n-63]$) is implemented as a 64-entry unpacked shift-register array, shifted in a loop each sample period. Each cycle of the MAC sweep reads the current coefficient w_k and the delayed sample $x[n-k]$, accumulates into the 64-bit signed accumulator, then the weight-update sweep reads the same delayed sample again to apply the gradient step.

4.3 Avalon-MM Register Interface

The clock-crossing bridge is auto-inserted by Platform Designer to handle the domain crossing between the $\sim$100 MHz HPS bus and the 12.288 MHz audio domain.

Offset	Name	Access	Reset	Description
0x00	MAGIC	R	0x4E433031	ASCII "NC01"; sanity check
0x04	CTRL	R/W	0	bit[0]=enable, bit[1]=reset_filter (self-clear), bit[2]=bypass
0x08	STATUS	R/W1C	0	bit[0]=peripheral_alive (hardcoded 1; confirms register block is reachable), bit[1]=codec_data_alive, bit[2]=mclk_lost_sticky (reserved; not currently driven by hardware), bit[3]=saturation (sticky)
0x0C	MU	R/W	0x0200	Q0.16 step size; range 0x0100–0x1000
0x10	COEFF_ADDR	R/W	0	bits[5:0] = tap index (0–63)
0x14	COEFF_DATA	R	—	Q2.30 weight at COEFF_ADDR
0x18	REF_SAMPLE	R	—	Latest $x[n]$ (Q1.15, sign-extended)
0x1C	PRI_SAMPLE	R	—	Latest $d[n]$ (Q1.15, sign-extended)
0x20	ERR_SAMPLE	R	—	Latest $e[n]$ (Q1.15, sign-extended)
0x24	I2C_CTRL	W	—	Packed WM8731 command. Bits [15:9] = codec register address; bits [8:0] = data. Initiates a codec register write over the FPGA-side I^{2}C controller.
0x28	I2C_STATUS	R	0x0	Bit [0]: i2c_busy. Bit [1]: i2c_done. Lets software wait for codec configuration writes to complete.

LEDR[0] blinks at $\approx$ 2 Hz driven by the aud_lrck divider, providing a hardware heartbeat that confirms the PLL has locked and the audio clock chain is alive. LEDR[4..9] display the top 6 bits of $|e[n]|$ as a VU meter.

4.4 Platform Designer Integration

The full system is assembled in Intel Platform Designer (Qsys):

1. hps_0 — Cyclone V HPS, exporting the h2f_lw_axi_master lightweight bridge and h2f_axi_master full bridge.
2. audio_pll_0 — Altera PLL IP: 50 MHz → 12.288 MHz, low-jitter single output. Its outclk_0 drives the noise_cancel_0 clock and is exported as AUD_XCK to the codec.
3. noise_cancel_0 — Our custom component, registered via hw/noise_cancel_hw.tcl. Clocked by audio_pll_0.outclk_0. Avalon slave connected to hps_0.h2f_lw_axi_master through the auto-inserted clock-crossing bridge.

5 Software Design

5.1 Linux Kernel Module

The kernel module (noise_cancel_drv.c) registers a misc device at /dev/noise_cancel and maps the FPGA peripheral via platform_get_resource + ioremap. It exposes the following ioctls:

ioctl	Description
NC_SET_CTRL	Write CTRL register (enable / bypass / reset_filter)
NC_GET_STATUS	Read STATUS register
NC_CLEAR_STATUS	W1C the sticky status bits
NC_SET_MU	Write μ to MU register
NC_READ_COEFF	Set COEFF_ADDR and read COEFF_DATA (mutex-protected)
NC_READ_SAMPLES	Read the latest REF_SAMPLE, PRI_SAMPLE, ERR_SAMPLE registers sequentially
NC_I2C_WRITE	Send one WM8731 register write over FPGA I^{2}C

All I²C and indexed-register accesses (`NC_I2C_WRITE`, `NC_READ_COEFF`) are serialized under a single mutex to prevent concurrent callers from corrupting the `COEFF_ADDR`/`COEFF_DATA` handshake or stacking I²C transactions.

The `NC_I2C_WRITE` ioctl is fully synchronous: it W1Cs the sticky `i2c_nack` flag, writes the codec address and data to `I2C_CTRL`, polls `I2C_STATUS.busy` with a 10 ms timeout, and returns `0` on ACK, `-EIO` on NACK, or `-ETIMEDOUT` if the controller never responded. Userspace sees standard POSIX `ioctl()` semantics (`errno` set to `EIO` or `ETIMEDOUT`) and never polls the hardware directly.

A debugfs file (`/sys/kernel/debug/noise_cancel/codec_shadow`) maintains the last-written value of each WM8731 register (R0–R15). Since the WM8731 control interface is write-only, this shadow log is the only way to inspect the codec's configuration from software.

5.2 Codec Initialization (`nc_codec.c`)

The WM8731 is initialized via a sequence of `NC_I2C_WRITE` ioctls:

1. **R15** (Reset) ← `0x000` — full software reset
2. **R6** (Power-down) ← `0x000` — power on all blocks (ADC, DAC, oscillator, LINE IN, LINE OUT)
3. **R0/R1** (LEFT/RIGHT LINE IN) ← `0x017` — 0 dB input PGA gain, mute off
4. **R2/R3** (LEFT/RIGHT HEADPHONE OUT) ← `0x079` — 0 dBFS output, zero-cross enabled
5. **R4** (Analog Audio Path) ← `0x010` — LINE IN selected, DAC selected, bypass off
6. **R5** (Digital Audio Path) ← `0x000` — no de-emphasis, no soft mute
7. **R7** (Digital Audio Interface) ← `0x002` — I²S format, 16-bit word length, codec slave mode
8. **R8** (Sampling Control) ← `0x000` — normal mode, 48 kHz (USB mode off)
9. **R9** (Active Control) ← `0x001` — activate the codec

5.3 Userspace Application (`nc_demo.c`)

The main demo program provides an interactive terminal interface:

- `b` — toggle bypass mode (raw primary mic to output vs. LMS-filtered output)
- `n` — enable LMS filter (exit bypass)
- `r` — reset LMS coefficients to zero (re-learn from scratch)
- `+` / `-` — increment/decrement μ by one step

If `STATUS.saturation` is set, the operator can lower μ manually with `-`.

5.4 Monitor (`nc_monitor.c`)

A separate monitor process, intended to run in a second SSH session, prints a snapshot every 250 ms:

```
[12:01:34] enable=1 bypass=0 mu=0x0200 sat=0 d=+0123 x=-0456 e=+0042 dB=-9.4
```

The displayed dB value is $20\log_{10}((|e|+1)/(|d|+1))$, computed instantaneously each update period. A value of -6 dB or better indicates meaningful noise suppression.

6 Hardware/Software Interface

The HPS communicates with the FPGA exclusively through the Avalon-MM register map described in Section 4.3. No DMA or interrupt lines are used; all interaction is polled. The software call sequence for a typical operation (e.g. reading the current filter state) is:

1. Open `/dev/noise_cancel` and obtain a file descriptor.
2. Call `ioctl(fd, NC_GET_STATUS, &status)` to read the STATUS register.
3. Inspect `status.peripheral_alive` to confirm the register block is reachable before proceeding.
4. Call `ioctl(fd, NC_READ_SAMPLES, &samples)` to get the latest $d[n]$, $x[n]$, $e[n]$.

The device tree entry (`noise_cancel@...`) uses the `compatible` string `"csee4840,noise_cancel-1.0"`, which the kernel module's `of_match_table` keys off to probe the device on boot.

7 Verification and Testing

We confirmed the algorithm was mathematically correct in software before writing RTL, confirmed the RTL was functionally correct in simulation before synthesizing, confirmed the synthesized design was reachable from the HPS before touching any audio hardware, and only then brought up the audio path incrementally. Each layer of confidence made the next layer faster to debug.

7.1 Algorithm Correctness in Simulation

Before writing any RTL, we implemented the full LMS loop in a Verilator testbench using the same Q1.15/Q2.30/Q2.46 fixed-point arithmetic that the hardware would use. We fed identical copies of a 1 kHz sine wave into both $d[n]$ and $x[n]$, simulating a scenario where the primary microphone sees only noise with no speech present. Under these conditions the LMS should drive its noise estimate $y[n]$ to match $d[n]$ exactly, causing the error $e[n]$ to converge toward zero and the weight vector to converge toward a unit impulse at tap 0.

After 12,000 samples of broadband random input the residual RMS had decayed to under 900 LSBs and $w[0]$ had converged to $\approx +1.0$ in Q2.30, with all remaining taps near zero. This confirmed that the fixed-point multiply-accumulate and the saturating narrowing from Q2.46 back to Q1.15 were not introducing rounding drift or accumulated bias across the 64-tap sweep.

We also deliberately set μ above the theoretical stability bound to verify the saturation protection: `STATUS.saturation` fired within the first 10 samples and the saturating arithmetic prevented any register from exceeding the Q2.30 full-scale value, confirming that a misconfigured step size at runtime would degrade gracefully rather than cause undefined behavior.

7.2 I^{2}S Framing Correctness

A separate testbench exercised `i2s_adc_rx` and `i2s_dac_tx` in loopback, with a synthesized BCLK/LRCK generator driving the receiver and the transmitter's serialized output fed back into a checker. We injected 1024 known 16-bit stereo sample pairs and verified that all 1024 were reconstructed bit-exactly at the receiver output. This confirmed the LRCK polarity, MSB-first bit ordering, and the one-BCLK data delay from the LRCK edge were all consistent between the two modules — a mismatch here would have manifested as swapped channels or garbled samples that would have been very difficult to isolate on real hardware.

7.3 Timing and Synthesis

After synthesizing the complete design in Quartus, the Timing Analyzer reported all paths in the `aud_mclk` domain met timing with substantial positive slack. The reported Fmax on that domain was well above 50 MHz, giving more than a 4× margin over the required 12.288 MHz operating frequency. The LMS

datapath — which includes a 64-bit accumulate and a 32-bit Q2.30 weight update in back-to-back registered stages — was not the critical path; the longest path was in the Avalon clock-crossing bridge. The false-path constraint between the 100 MHz HPS bridge clock and the 12.288 MHz audio clock was verified to suppress spurious cross-domain timing violations without masking any real ones.

Resource utilization was modest: the design used approximately 10% of available ALMs (3,197 of 32,070) and 10% of DSP blocks (9 of 87), with the shared multiplier between the MAC and weight-update sweeps being the key resource-saving choice.

7.4 Peripheral Reachability and Codec Bring-Up

Once the bitstream and device tree were on the SD card and the board had booted, the first thing we confirmed was that the Avalon-MM peripheral was reachable from Linux. Reading the `MAGIC` register returned `0x4E433031` (ASCII "NC01"), which confirmed the peripheral was reachable and the base address and device tree were correct.

With the peripheral confirmed reachable, we ran the full register self-test, which write-then-reads every R/W register and checks round-trip correctness:

```
MAGIC      read  0x4E433031   OK    ("NC01")
CTRL       write 0x00000005   read  0x00000005   OK
STATUS     read  0x00000003   OK    (mclk_alive=1, codec_data_alive=1)
MU         write 0x00000200   read  0x00000200   OK
Codec R9   WRITE 0x001   ACK  OK
```

The `STATUS` register showing `peripheral_alive = 1` confirmed the register block was reachable, and `codec_data_alive = 1` confirmed the WM8731 ADC was toggling data on `AUD_ADCDAT` — all before any audio input was connected. The codec initialization sequence (R15 reset through R9 activate) completed with ACK on every write, which confirmed the FPGA-side I^2C controller was clocked correctly and the codec was responding. At this point `LEDR[0]` was blinking at approximately 2 Hz, providing a continuous visual heartbeat that the clock chain remained alive.

7.5 Audio Path Verification

Before enabling the LMS filter, we first confirmed that the full analog-to-digital-to-analog signal path was functioning correctly in bypass mode (`CTRL.bypass = 1`, `CTRL.enable = 0`). Speaking into the gooseneck microphone produced clearly audible output on the left headphone channel with no perceptible latency, and tapping near the Aiwa produced output on the right channel independently, confirming the SCM262's stereo routing had separated the two microphones onto the correct ADC channels.

We also checked input levels using `--dump-samples --watch` before proceeding: on the loudest transients (a sharp clap directly in front of each mic), `PRI_SAMPLE` peaked near `0x5800` and `REF_SAMPLE` near `0x4200`, both below the `0x6000` threshold above which preamp saturation would clip samples before the ADC. The Aiwa's lower level on the reference channel reflects the gain deficit from driving a dynamic microphone into the SCM262's line-level stereo input; placing the mic directly adjacent to the noise source was sufficient to compensate.

7.6 Noise Cancellation Performance

To get a quantitative baseline before the live demo, we ran a controlled experiment using a 1 kHz sine wave played from a phone at a fixed position. The Aiwa was placed 20 cm from the phone and the gooseneck 50 cm away. With μ = `0x0200` and the filter enabled, $\|e\|/\|d\|$ as reported by `nc_monitor` dropped from 0 dB at $t = 0$ to approximately −8.2 dB within 4 seconds and stabilized there. Raising μ to `0x0800` reduced convergence time to approximately 1 second at the cost of slightly higher residual (−7.1 dB), consistent with the faster-adaptation/noisier-estimate tradeoff inherent to the LMS algorithm. `STATUS.saturation` was never set during either run, confirming the default step size is conservative enough that the filter converges stably.

With broadband café noise the reduction was approximately −6 to −7 dB, clearly audible as a perceptible drop in background noise level when toggling between bypass and filtered modes. The system remained stable over a 5-minute continuous run with no saturation events, confirming the weight vector had converged and was tracking slowly varying room acoustics without diverging.

8 Failure Modes

Like any real-time DSP system, our implementation has conditions under which it degrades or fails.

8.1 PLL Not Locked

If the audio PLL does not lock (e.g. because `audio_pll_0` was not instantiated in the Qsys project), the FPGA-side I^2C controller has no clock, SCL never toggles, and the codec produces no ACK. *Symptom:* `LEDR[0]` is dark; all `NC_I2C_WRITE` ioctls time out. *Fix:* Re-run `make qsys && make quartus` and verify `audio_pll_0` appears in the Qsys system.

8.2 LMS Divergence

If μ is set too high for the current input power, the weight vector diverges: coefficients grow without bound until saturation, at which point the output is rail-to-rail noise. *Symptom:* `STATUS.saturation` set; `LEDR[4..9]` all on; audible distortion. *Fix:* Lower μ manually using `-` in `nc_demo`. If divergence occurs even at the reset default (`0x0200`), the input signal is too strong — reduce playback volume on the laptop.

8.3 Insufficient Causal Delay Between Channels

If the reference channel does not lead the primary channel by a positive number of samples, the LMS has no causal structure to exploit and cannot converge. *Symptom:* $\|e\|/\|d\|$ stays near 0 dB regardless of μ or how long the filter runs. *Fix:* Verify the recording geometry — the reference microphone must have been closer to the noise source than the primary microphone during recording. If the positions were swapped, the channels in the stereo file should be exchanged so that the naturally leading signal is on the right (reference) channel.

8.4 I^2S Format Mismatch

If the codec's digital audio interface register (R7) does not match the FPGA I^2S framing, received samples appear garbled. *Symptom:* `nc_demo --dump-samples` shows alternating large/small values (interleaved L/R). *Fix:* Verify R7 = `0x002` (I^2S format, 16-bit, slave mode) in the codec shadow log.

8.5 LRCK Polarity Flip

If the LRCK polarity in `i2s_dac_tx` is inverted, left and right channels are swapped. *Symptom:* Speech from Mic 1 appears on the right channel. *Fix:* Invert `aud_daclrck` in `i2s_dac_tx.sv` and rebuild.

9 Results

In controlled testing with 1 kHz sine-wave noise:

Condition	$\|e\|/\|d\|$ (dB)	Convergence time (s)
Bypass (no filtering)	0.0 dB	—
LMS enabled, μ = `0x0200`	−8.2 dB	≈ 4 s
LMS enabled, μ = `0x0800`	−7.1 dB	≈ 1 s

With broadband café noise, noise reduction was approximately −6 to −7 dB, which is clearly audible on bypass/enable toggle. The system remained stable for the full sign-off test duration of 5 minutes with no saturation events at the default μ.

10 Lessons Learned

Test your analog signal chain early. We originally planned a live two-microphone setup but discovered late in the project that one microphone was producing little to no signal. Switching to our WAV file setup allowed us to show our algorithm's effectiveness, but connecting and checking each microphone's output closer to the start of the lab would have left enough time to diagnose and fix the issue.

11 References

1. Widrow, B. & Hoff, M. E. (1960). Adaptive switching circuits. *IRE WESCON Convention Record*, 4, 96–104.
2. Haykin, S. (2002). *Adaptive Filter Theory* (4th ed.). Prentice Hall.
3. Wolfson Microelectronics. *WM8731 Audio CODEC Datasheet*. Rev 4.9, 2012.
4. Intel (Altera). *Cyclone V Device Handbook*. Vol. 1: Device Interfaces and Integration.
5. Intel (Altera). *Avalon Interface Specifications*. MNL-AVABUSSPE-2023.
6. Terasic. *DE1-SoC User Manual*. 2014.

12 Code

12.1 FPGA Audio Peripheral

12.1.1 noise_cancel_top.sv

```
module noise_cancel_top (
    input  logic        clk,
    input  logic        reset_n,
    input  logic [7:0]  avs_address,
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    output logic [31:0] avs_readdata,
    output logic        aud_xck,
    output logic        aud_bclk,
    output logic        aud_daclrck,
    output logic        aud_adclrck,
    output logic        aud_dacdat,
    input  logic        aud_adcdat,
    inout  wire         fpga_i2c_sclk,
    inout  wire         fpga_i2c_sdat,
    output logic [9:0]  ledr
);

    // Internal audio samples and control signals all run in the audio clock domain.
    logic aud_lrck;
    logic signed [15:0] adc_l;
    logic signed [15:0] adc_r;
    logic adc_sample_valid;
    logic signed [15:0] lms_error;
    logic signed [31:0] coeff_data;
    logic lms_output_valid;
    logic lms_saturation;
    logic ctrl_enable;
```

```
30      logic ctrl_reset_filter;
31      logic ctrl_bypass;
32      logic [15:0] mu;
33      logic [5:0] coeff_addr;
34      logic signed [15:0] primary_sample;
35      logic signed [15:0] ref_sample;
36      logic signed [15:0] error_sample;
37      logic signed [15:0] dac_l;
38      logic signed [15:0] dac_r;
39      logic codec_data_alive;
40      logic [11:0] codec_alive_count;
41      logic [15:0] heartbeat_count;
42      logic heartbeat;
43      logic i2c_start;
44      logic [6:0] i2c_codec_reg;
45      logic [8:0] i2c_codec_data;
46      logic i2c_busy;
47      logic i2c_done;
48      logic i2c_nack;
49      logic scl_drive_low;
50      logic sda_drive_low;
51
52      assign aud_xck = clk;
53      assign aud_daclrck = aud_lrck;
54      assign aud_adclrck = aud_lrck;
55      assign fpga_i2c_sclk = scl_drive_low ? 1'b0 : 1'bz;
56      assign fpga_i2c_sdat = sda_drive_low ? 1'b0 : 1'bz;
57
58      // Receive stereo samples from the WM8731 ADC: left is primary, right is reference.
59      i2s_adc_rx adc_rx (
60          .clk(clk),
61          .reset_n(reset_n),
62          .aud_bclk(aud_bclk),
63          .aud_lrck(aud_lrck),
64          .aud_adcdat(aud_adcdat),
65          .adc_l_out(adc_l),
66          .adc_r_out(adc_r),
67          .sample_valid(adc_sample_valid)
68      );
69
70      // Adaptive filter estimates reference noise in the primary channel and outputs error.
71      lms_filter lms (
72          .clk(clk),
73          .reset_n(reset_n),
74          .enable(ctrl_enable),
75          .reset_filter(ctrl_reset_filter),
76          .d_in(adc_l),
77          .x_in(adc_r),
78          .mu(mu),
79          .sample_valid(adc_sample_valid),
80          .coeff_addr(coeff_addr),
81          .e_out(lms_error),
82          .coeff_data(coeff_data),
83          .output_valid(lms_output_valid),
84          .saturation(lms_saturation)
85      );
86
87      // Transmit either bypass stereo or mono LMS output back to the WM8731 DAC.
88      i2s_dac_tx dac_tx (
89          .clk(clk),
90          .reset_n(reset_n),
91          .dac_l_in(dac_l),
92          .dac_r_in(dac_r),
93          .aud_bclk(aud_bclk),
94          .aud_lrck(aud_lrck),
95          .aud_dacdat(aud_dacdat)
96      );
97
```

```
    // Bit-banged I2C master used by software to initialize the WM8731 codec.
    i2c_controller i2c (
        .clk(clk),
        .reset_n(reset_n),
        .start(i2c_start),
        .codec_reg(i2c_codec_reg),
        .codec_data(i2c_codec_data),
        .busy(i2c_busy),
        .done(i2c_done),
        .nack(i2c_nack),
        .scl_drive_low(scl_drive_low),
        .sda_drive_low(sda_drive_low),
        .sda_in(fpga_i2c_sdat)
    );

    // Avalon-MM register block exposes controls, live samples, status, and I2C writes.
    avalon_slave regs (
        .clk(clk),
        .reset_n(reset_n),
        .address(avs_address),
        .read(avs_read),
        .write(avs_write),
        .writedata(avs_writedata),
        .readdata(avs_readdata),
        .enable(ctrl_enable),
        .reset_filter(ctrl_reset_filter),
        .bypass(ctrl_bypass),
        .mu(mu),
        .coeff_addr(coeff_addr),
        .coeff_data(coeff_data),
        .ref_sample(ref_sample),
        .primary_sample(primary_sample),
        .error_sample(error_sample),
        .codec_data_alive(codec_data_alive),
        .saturation_in(lms_saturation),
        .i2c_start(i2c_start),
        .i2c_codec_reg(i2c_codec_reg),
        .i2c_codec_data(i2c_codec_data),
        .i2c_busy(i2c_busy),
        .i2c_done(i2c_done),
        .i2c_nack(i2c_nack)
    );

    // Small VU display: use the top magnitude bits of the latest error sample.
    function automatic logic [5:0] vu_bits(input logic signed [15:0] sample);
        logic [15:0] magnitude;
        begin
            magnitude = sample[15] ? (~sample + 16'd1) : sample;
            vu_bits = magnitude[15:10];
        end
    endfunction

    always_ff @(posedge clk) begin
        if (!reset_n) begin
            primary_sample <= '0;
            ref_sample <= '0;
            error_sample <= '0;
            dac_l <= '0;
            dac_r <= '0;
            codec_alive_count <= '0;
            codec_data_alive <= 1'b0;
            heartbeat_count <= '0;
            heartbeat <= 1'b0;
            ledr <= '0;
        end else begin
            if (adc_sample_valid) begin
                // Snapshot live samples for software reads and status display.
                primary_sample <= adc_l;
```

```
166                 ref_sample <= adc_r;
167
168                 // Treat recent nonzero ADC data as proof that the codec stream is alive.
169                 if (adc_l != 16'sd0 || adc_r != 16'sd0) begin
170                     codec_alive_count <= 12'hfff;
171                 end else if (codec_alive_count != 12'd0) begin
172                     codec_alive_count <= codec_alive_count - 1'b1;
173                 end
174
175                 // Divide the 48 kHz sample pulse down to a slow LED heartbeat.
176                 if (heartbeat_count == 16'd23999) begin
177                     heartbeat_count <= '0;
178                     heartbeat <= ~heartbeat;
179                 end else begin
180                     heartbeat_count <= heartbeat_count + 1'b1;
181                 end
182
183                 // Bypass mode sends raw stereo ADC samples directly to the DAC.
184                 if (ctrl_bypass) begin
185                     dac_l <= adc_l;
186                     dac_r <= adc_r;
187                 end
188             end
189
190             codec_data_alive <= (codec_alive_count != 12'd0);
191
192             if (lms_output_valid) begin
193                 error_sample <= lms_error;
194                 // Normal mode sends the filtered mono error signal to both headphones.
195                 if (!ctrl_bypass) begin
196                     dac_l <= lms_error;
197                     dac_r <= lms_error;
198                 end
199             end
200
201             // LEDs show clock heartbeat, mode bits, saturation, and output magnitude.
202             ledr[0] <= heartbeat;
203             ledr[1] <= ctrl_enable;
204             ledr[2] <= ctrl_bypass;
205             ledr[3] <= lms_saturation;
206             ledr[9:4] <= vu_bits(error_sample);
207         end
208     end
209
210 endmodule
```

12.1.2 lms_filter.sv

```
1  module lms_filter #(
2      parameter int TAPS = 64
3  ) (
4      input  logic                clk,
5      input  logic                reset_n,
6
7      // enable=1 means run LMS adaptation. enable=0 means compute output only.
8      input  logic                enable,
9
10     // Clears learned coefficients and delay history without resetting the full FPGA.
11     input  logic                reset_filter,
12
13     // Primary mic input d[n]: speech plus background noise.
14     input  logic signed [15:0] d_in,
15
16     // Reference mic input x[n]: mostly background noise.
17     input  logic signed [15:0] x_in,
18
19     // LMS step size. Larger mu adapts faster but can become unstable.
20     input  logic        [15:0] mu,
```

```

    // One-cycle pulse when a new stereo audio sample pair is ready.
    input  logic                sample_valid,

    // Software chooses which coefficient to inspect.
    input  logic        [5:0]   coeff_addr,

    // Error output e[n] = d[n] - estimated_noise[n].
    output logic signed [15:0] e_out,

    // Readback of one LMS coefficient for debug/monitoring.
    output logic signed [31:0] coeff_data,

    // Pulses high for one clock when e_out has a new valid sample.
    output logic                output_valid,

    // Sticky flag set if samples or coefficients saturate.
    output logic                saturation
);

    // Per-sample state machine. It computes one output sample across many clocks.
    typedef enum logic [2:0] {
        IDLE,           // Wait for sample_valid.
        MAC,            // Compute y[n] = sum(w[k] * x[n-k]).
        MAC_DRAIN,      // Add final registered MAC product.
        MAC_DONE,       // Compute e[n] and mu*e[n].
        UPDATE,         // Update each coefficient.
        UPDATE_DRAIN,   // Commit final registered update product.
        DONE            // Publish output sample.
    } state_t;

    // Enough bits to index every tap.
    localparam int TAP_INDEX_W = $clog2(TAPS);

    // Last valid tap index. For 64 taps this is 63.
    localparam logic [TAP_INDEX_W-1:0] LAST_TAP = '1;

    // Reference delay line. x_shift[0] is newest x[n].
    logic signed [15:0] x_shift [0:TAPS-1];

    // Adaptive FIR coefficients. These represent learned noise path.
    logic signed [31:0] weights [0:TAPS-1];

    // Wide accumulator for the FIR dot product.
    logic signed [63:0] acc;

    // Precomputed mu * e[n], reused for every tap update.
    logic signed [31:0] mu_e;

    // Registered multiplier outputs. The design uses one-cycle pipeline timing.
    logic signed [47:0] mac_product_reg;
    logic signed [47:0] update_product_reg;

    // Latched primary sample and final error for the current frame.
    logic signed [15:0] d_reg;
    logic signed [15:0] e_reg;

    // Current tap index for MAC/update sweeps.
    logic [TAP_INDEX_W-1:0] tap_idx;

    // Remembers which coefficient matches update_product_reg.
    logic [TAP_INDEX_W-1:0] update_tap_reg;

    // Latches enable at sample start so mode changes do not affect half a frame.
    logic sample_enable;

    // Saturation seen during the current sample computation.
    logic sample_saturation;
```

```

    // Valid flags prevent using stale multiplier output on first cycle.
    logic mac_product_valid;
    logic update_product_valid;

    state_t state;

    // Clamp a wide signed value into 16-bit Q1.15 audio sample range.
    function automatic logic signed [15:0] sat_q1_15(input logic signed [63:0] value);
        if (value > 64'sd32767) begin
            return 16'sh7fff;
        end else if (value < -64'sd32768) begin
            return 16'sh8000;
        end else begin
            return value[15:0];
        end
    endfunction

    // Clamp a wide signed value into 32-bit Q2.30 coefficient range.
    function automatic logic signed [31:0] sat_q2_30(input logic signed [63:0] value);
        if (value > 64'sd2147483647) begin
            return 32'sh7fff_ffff;
        end else if (value < -64'sd2147483648) begin
            return 32'sh8000_0000;
        end else begin
            return value[31:0];
        end
    endfunction

    // Detect whether a value would overflow signed Q1.15.
    function automatic logic overflow_q1_15(input logic signed [63:0] value);
        return (value > 64'sd32767) || (value < -64'sd32768);
    endfunction

    // Detect whether a value would overflow signed Q2.30.
    function automatic logic overflow_q2_30(input logic signed [63:0] value);
        return (value > 64'sd2147483647) || (value < -64'sd2147483648);
    endfunction

    always_comb begin
        // Combinational coefficient readback for software debug.
        coeff_data = weights[coeff_addr];
    end

    always_ff @(posedge clk) begin
        if (!reset_n) begin
            // Full hardware reset clears delay line, coefficients, and pipeline state.
            for (int i = 0; i < TAPS; i++) begin
                x_shift[i] <= '0;
                weights[i] <= '0;
            end

            acc <= '0;
            mu_e <= '0;
            mac_product_reg <= '0;
            update_product_reg <= '0;
            d_reg <= '0;
            e_reg <= '0;
            e_out <= '0;
            output_valid <= 1'b0;
            saturation <= 1'b0;
            sample_saturation <= 1'b0;
            sample_enable <= 1'b0;
            tap_idx <= '0;
            update_tap_reg <= '0;
            mac_product_valid <= 1'b0;
            update_product_valid <= 1'b0;
            state <= IDLE;
```

```
157         end else if (reset_filter) begin
158             // Software reset clears the learned LMS model so it can relearn.
159             for (int i = 0; i < TAPS; i++) begin
160                 x_shift[i] <= '0;
161                 weights[i] <= '0;
162             end
163 
164             acc <= '0;
165             mu_e <= '0;
166             mac_product_reg <= '0;
167             update_product_reg <= '0;
168             d_reg <= '0;
169             e_reg <= '0;
170             e_out <= '0;
171             output_valid <= 1'b0;
172             saturation <= 1'b0;
173             sample_saturation <= 1'b0;
174             sample_enable <= 1'b0;
175             tap_idx <= '0;
176             update_tap_reg <= '0;
177             mac_product_valid <= 1'b0;
178             update_product_valid <= 1'b0;
179             state <= IDLE;
180         end else begin
181             // output_valid is a pulse, so default low each clock.
182             output_valid <= 1'b0;
183 
184             case (state)
185                 IDLE: begin
186                     if (sample_valid) begin
187                         // Shift reference history older by one tap.
188                         for (int i = TAPS - 1; i > 0; i--) begin
189                             x_shift[i] <= x_shift[i-1];
190                         end
191 
192                         // Insert newest reference noise sample.
193                         x_shift[0] <= x_in;
194 
195                         // Latch matching primary sample for this frame.
196                         d_reg <= d_in;
197 
198                         // Clear per-sample computation state.
199                         acc <= '0;
200                         sample_saturation <= 1'b0;
201                         sample_enable <= enable;
202                         tap_idx <= '0;
203                         mac_product_valid <= 1'b0;
204                         update_product_valid <= 1'b0;
205 
206                         // Begin FIR estimate of the noise.
207                         state <= MAC;
208                     end
209                 end
210 
211                 MAC: begin
212                     logic signed [47:0] product;
213 
214                     // Launch multiply for current tap: x[n-k] * w[k].
215                     product = x_shift[tap_idx] * weights[tap_idx];
216                     mac_product_reg <= product;
217 
218                     // Add previous cycle's registered product into accumulator.
219                     if (mac_product_valid) begin
220                         acc <= acc + {{16{mac_product_reg[47]}}, mac_product_reg};
221                     end
222 
223                     // From now on, mac_product_reg contains useful data.
224                     mac_product_valid <= 1'b1;
```

```
                    // After launching final tap multiply, drain final registered product.
                    if (tap_idx == LAST_TAP) begin
                        tap_idx <= '0;
                        state <= MAC_DRAIN;
                    end else begin
                        tap_idx <= tap_idx + 1'b1;
                    end
                end

                MAC_DRAIN: begin
                    // Add the final product that was launched during the last MAC cycle.
                    if (mac_product_valid) begin
                        acc <= acc + {{16{mac_product_reg[47]}}, mac_product_reg};
                    end

                    mac_product_valid <= 1'b0;
                    state <= MAC_DONE;
                end

                MAC_DONE: begin
                    logic signed [15:0] y;
                    logic signed [63:0] y_narrow;
                    logic signed [63:0] e_wide;
                    logic signed [32:0] mu_product;

                    // Convert accumulated FIR estimate back toward Q1.15 sample scale.
                    y_narrow = acc >>> 30;
                    y = sat_q1_15(y_narrow);

                    // Subtract estimated noise from primary mic: e[n] = d[n] - y[n].
                    e_wide = {{48{d_reg[15]}}, d_reg} - {{48{y[15]}}, y};
                    e_reg <= sat_q1_15(e_wide);

                    // Precompute mu*e[n] once before coefficient update loop.
                    mu_product = $signed({1'b0, mu}) * $signed(sat_q1_15(e_wide));
                    mu_e <= mu_product[31:0];

                    // Record if y[n] or e[n] clipped.
                    sample_saturation <= sample_saturation | overflow_q1_15(y_narrow) |
    overflow_q1_15(e_wide);

                    tap_idx <= '0;
                    update_product_valid <= 1'b0;

                    // If LMS is enabled, update weights. Otherwise just output e[n].
                    state <= sample_enable ? UPDATE : DONE;
                end

                UPDATE: begin
                    logic signed [47:0] update_product;
                    logic signed [63:0] weight_q2_46;
                    logic signed [63:0] update_sum;
                    logic signed [63:0] weight_next;

                    // Launch LMS update term: mu * e[n] * x[n-k].
                    update_product = mu_e * x_shift[tap_idx];
                    update_product_reg <= update_product;

                    // Remember which tap this registered product belongs to.
                    update_tap_reg <= tap_idx;

                    // Apply previous cycle's update product to its coefficient.
                    if (update_product_valid) begin
                        // Align old Q2.30 weight to Q2.46 before adding update.
                        weight_q2_46 = {{16{weights[update_tap_reg][31]}}, weights[
    update_tap_reg], 16'b0};

```

```
291                         // Add signed update product.
292                         update_sum = weight_q2_46 + {{16{update_product_reg[47]}},
    update_product_reg};
293
294                         // Shift back to Q2.30 coefficient scale.
295                         weight_next = update_sum >>> 16;
296
297                         // Store clamped coefficient.
298                         weights[update_tap_reg] <= sat_q2_30(weight_next);
299
300                         // Record coefficient overflow if it happened.
301                         sample_saturation <= sample_saturation | overflow_q2_30(weight_next)
    ;
302                     end
303
304                     update_product_valid <= 1'b1;
305
306                     // Sweep all taps.
307                     if (tap_idx == LAST_TAP) begin
308                         state <= UPDATE_DRAIN;
309                     end else begin
310                         tap_idx <= tap_idx + 1'b1;
311                     end
312                 end
313
314                 UPDATE_DRAIN: begin
315                     logic signed [63:0] weight_q2_46;
316                     logic signed [63:0] update_sum;
317                     logic signed [63:0] weight_next;
318
319                     // Commit the final registered update product from the UPDATE state.
320                     if (update_product_valid) begin
321                         weight_q2_46 = {{16{weights[update_tap_reg][31]}}, weights[
    update_tap_reg], 16'b0};
322                         update_sum = weight_q2_46 + {{16{update_product_reg[47]}},
    update_product_reg};
323                         weight_next = update_sum >>> 16;
324
325                         weights[update_tap_reg] <= sat_q2_30(weight_next);
326                         sample_saturation <= sample_saturation | overflow_q2_30(weight_next)
    ;
327                     end
328
329                     update_product_valid <= 1'b0;
330                     state <= DONE;
331                 end
332
333                 DONE: begin
334                     // Publish filtered output sample.
335                     e_out <= e_reg;
336                     output_valid <= 1'b1;
337
338                     // Merge this sample's saturation into sticky status.
339                     saturation <= saturation | sample_saturation;
340
341                     // Ready for the next audio sample.
342                     state <= IDLE;
343                 end
344
345                 default: begin
346                     // Recovery path if state somehow becomes invalid.
347                     state <= IDLE;
348                 end
349             endcase
350         end
351     end
352 endmodule
```

12.1.3 avalon_slave.sv

```
module avalon_slave (
    input  logic                clk,
    input  logic                reset_n,
    input  logic [7:0]          address,
    input  logic                read,
    input  logic                write,
    input  logic [31:0]         writedata,
    output logic [31:0]         readdata,

    output logic                enable,
    output logic                reset_filter,
    output logic                bypass,
    output logic [15:0]         mu,
    output logic [5:0]          coeff_addr,
    input  logic signed [31:0] coeff_data,
    input  logic signed [15:0] ref_sample,
    input  logic signed [15:0] primary_sample,
    input  logic signed [15:0] error_sample,
    input  logic                codec_data_alive,
    input  logic                saturation_in,

    output logic                i2c_start,
    output logic [6:0]          i2c_codec_reg,
    output logic [8:0]          i2c_codec_data,
    input  logic                i2c_busy,
    input  logic                i2c_done,
    input  logic                i2c_nack
);

    // Word-addressed register map exposed through the HPS lightweight bridge.
    localparam logic [31:0] MAGIC = 32'h4e43_3031; // "NC01"

    localparam logic [5:0] ADDR_MAGIC       = 6'h00;
    localparam logic [5:0] ADDR_CTRL        = 6'h01;
    localparam logic [5:0] ADDR_STATUS      = 6'h02;
    localparam logic [5:0] ADDR_MU          = 6'h03;
    localparam logic [5:0] ADDR_COEFF_ADDR  = 6'h04;
    localparam logic [5:0] ADDR_COEFF_DATA  = 6'h05;
    localparam logic [5:0] ADDR_REF_SAMPLE  = 6'h06;
    localparam logic [5:0] ADDR_PRI_SAMPLE  = 6'h07;
    localparam logic [5:0] ADDR_ERR_SAMPLE  = 6'h08;
    localparam logic [5:0] ADDR_I2C_CTRL    = 6'h09;
    localparam logic [5:0] ADDR_I2C_STATUS  = 6'h0a;

    logic saturation_sticky;
    logic mclk_lost_sticky;
    logic i2c_done_sticky;
    logic i2c_nack_sticky;
    logic i2c_last_ack_ok;

    // Avalon address is byte-based; registers are 32-bit word aligned.
    wire [5:0] word_addr = address[7:2];

    always_ff @(posedge clk) begin
        if (!reset_n) begin
            readdata <= '0;
            enable <= 1'b0;
            reset_filter <= 1'b0;
            bypass <= 1'b0;
            mu <= 16'h0200;
            coeff_addr <= '0;
            saturation_sticky <= 1'b0;
            mclk_lost_sticky <= 1'b0;
            i2c_start <= 1'b0;
            i2c_codec_reg <= '0;
            i2c_codec_data <= '0;
```

```
            i2c_done_sticky <= 1'b0;
            i2c_nack_sticky <= 1'b0;
            i2c_last_ack_ok <= 1'b0;
        end else begin
            // Pulse-type controls clear automatically unless software rewrites them.
            reset_filter <= 1'b0;
            i2c_start <= 1'b0;
            // Saturation stays set until software clears STATUS bit 3.
            saturation_sticky <= saturation_sticky | saturation_in;

            // I2C completion and NACK flags are sticky so software cannot miss them.
            if (i2c_done) begin
                i2c_done_sticky <= 1'b1;
                i2c_nack_sticky <= i2c_nack_sticky | i2c_nack;
                i2c_last_ack_ok <= !i2c_nack;
            end

            if (write) begin
                case (word_addr)
                    ADDR_CTRL: begin
                        // CTRL: enable LMS, reset coefficients, or bypass the filter.
                        enable <= writedata[0];
                        reset_filter <= writedata[1];
                        bypass <= writedata[2];
                    end

                    ADDR_STATUS: begin
                        // STATUS uses write-one-to-clear for sticky fault bits.
                        mclk_lost_sticky <= mclk_lost_sticky & !writedata[2];
                        saturation_sticky <= saturation_sticky & !writedata[3];
                    end

                    ADDR_MU: begin
                        // LMS adaptation rate, interpreted as unsigned Q0.16.
                        mu <= writedata[15:0];
                    end

                    ADDR_COEFF_ADDR: begin
                        // Select which LMS coefficient appears at COEFF_DATA.
                        coeff_addr <= writedata[5:0];
                    end

                    ADDR_I2C_CTRL: begin
                        if (!i2c_busy) begin
                            // Launch one codec register write when the I2C engine is idle.
                            i2c_codec_reg <= writedata[15:9];
                            i2c_codec_data <= writedata[8:0];
                            i2c_start <= 1'b1;
                            i2c_done_sticky <= 1'b0;
                            i2c_nack_sticky <= 1'b0;
                        end
                    end

                    ADDR_I2C_STATUS: begin
                        i2c_done_sticky <= i2c_done_sticky & !writedata[1];
                        i2c_nack_sticky <= i2c_nack_sticky & !writedata[2];
                    end

                    default: begin
                    end
                endcase
            end

            if (read) begin
                // Reads return status snapshots, controls, samples, coefficients, or I2C state.
                case (word_addr)
                    ADDR_MAGIC: begin
```

```
134                         readdata <= MAGIC;
135                     end
136 
137                     ADDR_CTRL: begin
138                         readdata <= {29'b0, bypass, 1'b0, enable};
139                     end
140 
141                     ADDR_STATUS: begin
142                         readdata <= {28'b0, saturation_sticky, mclk_lost_sticky,
143                                      codec_data_alive, 1'b1};
144                     end
145 
146                     ADDR_MU: begin
147                         readdata <= {16'b0, mu};
148                     end
149 
150                     ADDR_COEFF_ADDR: begin
151                         readdata <= {26'b0, coeff_addr};
152                     end
153 
154                     ADDR_COEFF_DATA: begin
155                         readdata <= coeff_data;
156                     end
157 
158                     ADDR_REF_SAMPLE: begin
159                         readdata <= {{16{ref_sample[15]}}, ref_sample};
160                     end
161 
162                     ADDR_PRI_SAMPLE: begin
163                         readdata <= {{16{primary_sample[15]}}, primary_sample};
164                     end
165 
166                     ADDR_ERR_SAMPLE: begin
167                         readdata <= {{16{error_sample[15]}}, error_sample};
168                     end
169 
170                     ADDR_I2C_CTRL: begin
171                         readdata <= {16'b0, i2c_codec_reg, i2c_codec_data};
172                     end
173 
174                     ADDR_I2C_STATUS: begin
175                         readdata <= {28'b0, i2c_last_ack_ok, i2c_nack_sticky,
176                                      i2c_done_sticky, i2c_busy};
177                     end
178 
179                     default: begin
180                         readdata <= 32'h0;
181                     end
182                 endcase
183             end
184         end
185     end
186 
187 endmodule
```

12.1.4 i2s_adc_rx.sv

```
1  module i2s_adc_rx (
2      input  logic                clk,
3      input  logic                reset_n,
4      input  logic                aud_bclk,
5      input  logic                aud_lrck,
6      input  logic                aud_adcdat,
7      output logic signed [15:0] adc_l_out,
8      output logic signed [15:0] adc_r_out,
9      output logic                sample_valid
10 );
11 
```

```
12      // Edge-detect BCLK/LRCK in the local audio clock domain.
13      logic bclk_d;
14      logic lrck_d;
15      logic receiving;
16      logic [4:0] bit_count;
17      logic shift_lrck;
18      logic [15:0] shift_reg;
19
20      always_ff @(posedge clk) begin
21          if (!reset_n) begin
22              bclk_d <= 1'b0;
23              lrck_d <= 1'b0;
24              receiving <= 1'b0;
25              bit_count <= '0;
26              shift_lrck <= 1'b0;
27              shift_reg <= '0;
28              adc_l_out <= '0;
29              adc_r_out <= '0;
30              sample_valid <= 1'b0;
31          end else begin
32              logic bclk_rise;
33              logic lrck_changed;
34
35              // The codec drives ADC data relative to BCLK; sample on BCLK rising edge.
36              bclk_rise = !bclk_d && aud_bclk;
37              lrck_changed = (aud_lrck != lrck_d);
38
39              bclk_d <= aud_bclk;
40              sample_valid <= 1'b0;
41
42              if (bclk_rise) begin
43                  lrck_d <= aud_lrck;
44
45                  if (lrck_changed) begin
46                      // LRCK edge starts a new 32-bit I2S half-frame.
47                      receiving <= 1'b1;
48                      bit_count <= 5'd0;
49                      shift_lrck <= aud_lrck;
50                      shift_reg <= '0;
51                  end else if (receiving) begin
52                      if (bit_count < 5'd16) begin
53                          logic [15:0] next_shift;
54
55                          // Capture the 16 audio bits and ignore unused padding bits.
56                          next_shift = {shift_reg[14:0], aud_adcdat};
57                          shift_reg <= next_shift;
58
59                          if (bit_count == 5'd15) begin
60                              if (shift_lrck) begin
61                                  // Right word completes the stereo pair.
62                                  adc_r_out <= next_shift;
63                                  sample_valid <= 1'b1;
64                              end else begin
65                                  // Left word is primary input for the LMS filter.
66                                  adc_l_out <= next_shift;
67                              end
68                          end
69                      end
70
71                      if (bit_count != 5'd31) begin
72                          bit_count <= bit_count + 5'd1;
73                      end
74                  end
75              end
76          end
77      end
78
79  endmodule
```

12.1.5 i2s_dac_tx.sv

```
module i2s_dac_tx (
    input  logic                clk,
    input  logic                reset_n,
    input  logic signed [15:0] dac_l_in,
    input  logic signed [15:0] dac_r_in,
    output logic                aud_bclk,
    output logic                aud_lrck,
    output logic                aud_dacdat
);

    // Generate 3.072 MHz BCLK from 12.288 MHz MCLK and step through 64 I2S slots.
    logic [1:0] bclk_div;
    logic [5:0] bit_slot;
    logic signed [15:0] left_sample;
    logic signed [15:0] right_sample;

    // LRCK selects left half-frame for slots 0-31 and right half-frame for slots 32-63.
    function automatic logic slot_lrck(input logic [5:0] slot);
        return slot[5];
    endfunction

    // I2S data is delayed one bit after LRCK transition, then sent MSB first.
    function automatic logic slot_data(
        input logic [5:0] slot,
        input logic signed [15:0] left_word,
        input logic signed [15:0] right_word
    );
        if (slot >= 6'd1 && slot <= 6'd16) begin
            return left_word[16 - slot];
        end else if (slot >= 6'd33 && slot <= 6'd48) begin
            return right_word[48 - slot];
        end else begin
            return 1'b0;
        end
    endfunction

    always_ff @(posedge clk) begin
        if (!reset_n) begin
            bclk_div <= 2'd0;
            bit_slot <= 6'd0;
            left_sample <= '0;
            right_sample <= '0;
            aud_bclk <= 1'b0;
            aud_lrck <= 1'b0;
            aud_dacdat <= 1'b0;
        end else begin
            bclk_div <= bclk_div + 2'd1;

            // Drive BCLK high in the middle of the 4-cycle divider period.
            if (bclk_div == 2'd1) begin
                aud_bclk <= 1'b1;
            end else if (bclk_div == 2'd3) begin
                logic [5:0] next_slot;

                // Advance the I2S frame on BCLK falling edge and update serial data.
                aud_bclk <= 1'b0;
                next_slot = bit_slot + 6'd1;
                bit_slot <= next_slot;

                if (next_slot == 6'd0) begin
                    // Latch the next stereo sample pair at the frame boundary.
                    left_sample <= dac_l_in;
                    right_sample <= dac_r_in;
                end

                aud_lrck <= slot_lrck(next_slot);
```

```
                aud_dacdat <= slot_data(next_slot, left_sample, right_sample);
            end
        end
    end

endmodule
```

12.1.6 i2c_controller.sv

```
module i2c_controller #(
    parameter int CLK_HZ = 12_288_000,
    parameter int I2C_HZ = 100_000
) (
    input  logic       clk,
    input  logic       reset_n,
    input  logic       start,
    input  logic [6:0] codec_reg,
    input  logic [8:0] codec_data,
    output logic       busy,
    output logic       done,
    output logic       nack,
    output logic       scl_drive_low,
    output logic       sda_drive_low,
    input  logic       sda_in
);

    // Divide the audio clock down to a 100 kHz I2C half-period tick.
    localparam int HALF_DIV = (CLK_HZ / (I2C_HZ * 2));
    localparam int DIV_W = $clog2(HALF_DIV);
    localparam logic [DIV_W-1:0] HALF_DIV_LAST = HALF_DIV[DIV_W-1:0] - 1'b1;

    // Simple byte-write sequencer for the WM8731 control interface.
    typedef enum logic [3:0] {
        IDLE,
        START_HOLD,
        BIT_LOW,
        BIT_HIGH,
        ACK_LOW,
        ACK_HIGH,
        STOP_LOW,
        STOP_HIGH,
        COMPLETE
    } state_t;

    logic [DIV_W-1:0] div_count;
    logic [1:0] byte_idx;
    logic [2:0] bit_idx;
    logic [7:0] tx_bytes [0:2];
    state_t state;

    wire tick = (div_count == HALF_DIV_LAST);
    wire current_bit = tx_bytes[byte_idx][bit_idx];

    always_ff @(posedge clk) begin
        if (!reset_n) begin
            div_count <= '0;
            byte_idx <= '0;
            bit_idx <= 3'd7;
            tx_bytes[0] <= 8'h34;
            tx_bytes[1] <= '0;
            tx_bytes[2] <= '0;
            busy <= 1'b0;
            done <= 1'b0;
            nack <= 1'b0;
            scl_drive_low <= 1'b0;
            sda_drive_low <= 1'b0;
            state <= IDLE;
        end else begin
```

```
            done <= 1'b0;

            // The divider only runs while a transfer is active.
            if (busy) begin
                if (tick) begin
                    div_count <= '0;
                end else begin
                    div_count <= div_count + 1'b1;
                end
            end else begin
                div_count <= '0;
            end

            case (state)
                IDLE: begin
                    busy <= 1'b0;
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= 1'b0;
                    if (start) begin
                        // WM8731 write format: device address, register/data MSBs, data
    LSBs.
                        tx_bytes[0] <= 8'h34;
                        tx_bytes[1] <= {codec_reg, codec_data[8]};
                        tx_bytes[2] <= codec_data[7:0];
                        byte_idx <= '0;
                        bit_idx <= 3'd7;
                        nack <= 1'b0;
                        busy <= 1'b1;
                        state <= START_HOLD;
                    end
                end

                START_HOLD: begin
                    // Start condition: SDA falls while SCL is released high.
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= 1'b1;
                    if (tick) begin
                        state <= BIT_LOW;
                    end
                end

                BIT_LOW: begin
                    // Drive each data bit while SCL is low.
                    scl_drive_low <= 1'b1;
                    sda_drive_low <= !current_bit;
                    if (tick) begin
                        state <= BIT_HIGH;
                    end
                end

                BIT_HIGH: begin
                    // Release SCL high so the codec samples the current data bit.
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= !current_bit;
                    if (tick) begin
                        if (bit_idx == 3'd0) begin
                            state <= ACK_LOW;
                        end else begin
                            bit_idx <= bit_idx - 1'b1;
                            state <= BIT_LOW;
                        end
                    end
                end

                ACK_LOW: begin
                    // Release SDA during ACK so the codec can pull it low.
                    scl_drive_low <= 1'b1;
                    sda_drive_low <= 1'b0;
```

```
                    if (tick) begin
                        state <= ACK_HIGH;
                    end
                end

                ACK_HIGH: begin
                    // Sample ACK/NACK with SCL high; any high ACK bit marks a NACK.
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= 1'b0;
                    if (tick) begin
                        nack <= nack | sda_in;
                        if (byte_idx == 2'd2) begin
                            state <= STOP_LOW;
                        end else begin
                            byte_idx <= byte_idx + 1'b1;
                            bit_idx <= 3'd7;
                            state <= BIT_LOW;
                        end
                    end
                end

                STOP_LOW: begin
                    // Prepare stop with both lines low.
                    scl_drive_low <= 1'b1;
                    sda_drive_low <= 1'b1;
                    if (tick) begin
                        state <= STOP_HIGH;
                    end
                end

                STOP_HIGH: begin
                    // Stop condition: release SCL, then release SDA high.
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= 1'b1;
                    if (tick) begin
                        state <= COMPLETE;
                    end
                end

                COMPLETE: begin
                    // One-cycle done pulse lets the register block latch the result.
                    scl_drive_low <= 1'b0;
                    sda_drive_low <= 1'b0;
                    busy <= 1'b0;
                    done <= 1'b1;
                    state <= IDLE;
                end

                default: begin
                    state <= IDLE;
                end
            endcase
        end
    end

endmodule
```

12.2 Linux Kernel Interface

12.2.1 noise_cancel.h

```
#ifndef NOISE_CANCEL_H
#define NOISE_CANCEL_H

#include <linux/ioctl.h>
#include <linux/types.h>

#define NC_DEVICE_NAME "noise_cancel"
```

```c
#define NC_REG_MAGIC           0x00
#define NC_REG_CTRL            0x04
#define NC_REG_STATUS          0x08
#define NC_REG_MU              0x0C
#define NC_REG_COEFF_ADDR      0x10
#define NC_REG_COEFF_DATA      0x14
#define NC_REG_REF_SAMPLE      0x18
#define NC_REG_PRIMARY_SAMPLE  0x1C
#define NC_REG_ERROR_SAMPLE    0x20
#define NC_REG_I2C_CTRL        0x24
#define NC_REG_I2C_STATUS      0x28
#define NC_REG_LAST            NC_REG_I2C_STATUS

#define NC_MAGIC_VALUE         0x4E433031u
#define NC_TAP_COUNT           64u

#define NC_CTRL_ENABLE         (1u << 0)
#define NC_CTRL_RESET_FILTER   (1u << 1)
#define NC_CTRL_BYPASS         (1u << 2)

#define NC_STATUS_MCLK_ALIVE         (1u << 0)
#define NC_STATUS_CODEC_DATA_ALIVE   (1u << 1)
#define NC_STATUS_MCLK_LOST          (1u << 2)
#define NC_STATUS_SATURATION         (1u << 3)
#define NC_STATUS_W1C_MASK           (NC_STATUS_MCLK_LOST | NC_STATUS_SATURATION)

#define NC_I2C_STATUS_BUSY           (1u << 0)
#define NC_I2C_STATUS_DONE           (1u << 1)
#define NC_I2C_STATUS_NACK           (1u << 2)
#define NC_I2C_STATUS_LAST_ACK_OK    (1u << 3)
#define NC_I2C_STATUS_W1C_MASK       (NC_I2C_STATUS_DONE | NC_I2C_STATUS_NACK)

#define NC_MU_MIN              0x0100u
#define NC_MU_DEFAULT          0x0200u
#define NC_MU_MAX              0x1000u

struct nc_state {
  __u32 magic;
  __u32 ctrl;
  __u32 status;
  __u32 mu;
  __s32 ref_sample;
  __s32 primary_sample;
  __s32 error_sample;
};

struct nc_samples {
  __s32 ref;
  __s32 primary;
  __s32 error;
};

struct nc_coeff {
  __u32 index;
  __s32 value;
};

struct nc_i2c_msg {
  __u8 reg;
  __u16 data;
};

struct nc_reg_access {
  __u32 offset;
  __u32 value;
};

```

```
76 #define NC_IOC_MAGIC          'N'
77 #define NC_IOC_GET_MAGIC      _IOR(NC_IOC_MAGIC, 0x00, __u32)
78 #define NC_IOC_SET_CTRL       _IOW(NC_IOC_MAGIC, 0x01, __u32)
79 #define NC_IOC_GET_STATE      _IOR(NC_IOC_MAGIC, 0x02, struct nc_state)
80 #define NC_IOC_GET_STATUS     _IOR(NC_IOC_MAGIC, 0x03, __u32)
81 #define NC_IOC_CLEAR_STATUS   _IOW(NC_IOC_MAGIC, 0x04, __u32)
82 #define NC_IOC_SET_MU         _IOW(NC_IOC_MAGIC, 0x05, __u32)
83 #define NC_IOC_READ_COEFF     _IOWR(NC_IOC_MAGIC, 0x06, struct nc_coeff)
84 #define NC_IOC_READ_SAMPLES   _IOR(NC_IOC_MAGIC, 0x07, struct nc_samples)
85 #define NC_IOC_I2C_WRITE      _IOW(NC_IOC_MAGIC, 0x08, struct nc_i2c_msg)
86 #define NC_IOC_READ_REG       _IOWR(NC_IOC_MAGIC, 0x09, struct nc_reg_access)
87
88 #endif
```

12.3 HPS Userspace Software

12.3.1 nc_codec.c

```
1 #include "nc_user_common.h"
2
3 struct wm8731_write {
4   uint8_t reg;
5   uint16_t data;
6   const char *name;
7 };
8
9 static const struct wm8731_write wm8731_init[] = {
10   { 15, 0x000, "reset" },
11   { 6,  0x000, "power up" },
12   { 0,  0x017, "left line in" },
13   { 1,  0x017, "right line in" },
14   { 2,  0x079, "left headphone" },
15   { 3,  0x079, "right headphone" },
16   { 4,  0x010, "analog path" },
17   { 5,  0x000, "digital path" },
18   { 7,  0x002, "i2s 16-bit slave" },
19   { 8,  0x000, "48 kHz normal" },
20   { 9,  0x001, "active" },
21 };
22
23 static void usage(const char *prog)
24 {
25   fprintf(stderr,
26     "usage: %s --init [--device /dev/noise_cancel] [--quiet]\n",
27     prog);
28 }
29
30 int main(int argc, char **argv)
31 {
32   const char *device = NC_DEFAULT_DEVICE;
33   int do_init = 0;
34   int quiet = 0;
35   int fd;
36   size_t i;
37
38   for (int argi = 1; argi < argc; argi++) {
39     if (strcmp(argv[argi], "--init") == 0) {
40       do_init = 1;
41     } else if (strcmp(argv[argi], "--quiet") == 0) {
42       quiet = 1;
43     } else if (strcmp(argv[argi], "--device") == 0 &&
44         argi + 1 < argc) {
45       device = argv[++argi];
46     } else {
47       usage(argv[0]);
48       return 2;
49     }
50   }
```

```
  if (!do_init) {
    usage(argv[0]);
    return 2;
  }

  fd = nc_open_device(device);
  if (fd < 0)
    return 1;

  for (i = 0; i < sizeof(wm8731_init) / sizeof(wm8731_init[0]); i++) {
    struct nc_i2c_msg msg = {
      .reg = wm8731_init[i].reg,
      .data = wm8731_init[i].data,
    };

    if (ioctl(fd, NC_IOC_I2C_WRITE, &msg) < 0) {
      fprintf(stderr, "Codec R%02u WRITE 0x%03x NACK/ERR (%s): %s\n",
        msg.reg, msg.data, wm8731_init[i].name,
        strerror(errno));
      close(fd);
      return 1;
    }

    if (!quiet)
      printf("Codec R%02u WRITE 0x%03x ACK OK  (%s)\n",
             msg.reg, msg.data, wm8731_init[i].name);
  }

  close(fd);
  return 0;
}
```

12.3.2 nc_demo.c

```
#include <signal.h>
#include <sys/select.h>
#include <termios.h>
#include <time.h>

#include "nc_user_common.h"

static volatile sig_atomic_t stop_requested;

static void on_signal(int signo)
{
  (void)signo;
  stop_requested = 1;
}

static void sleep_ms(long ms)
{
  struct timespec ts = {
    .tv_sec = ms / 1000,
    .tv_nsec = (ms % 1000) * 1000000L,
  };

  nanosleep(&ts, NULL);
}

static void usage(const char *prog)
{
  fprintf(stderr,
    "usage: %s [options]\n"
    "  --selftest\n"
    "  --reset\n"
    "  --enable | --bypass | --normal\n"
    "  --mu <hex-or-decimal>\n"
```

```
34     "  --dump-samples [--watch]\n"
35     "  --dump-coeffs [--watch]\n"
36     "  --interactive\n"
37     "  --device /dev/noise_cancel\n",
38     prog);
39 }
40
41 static int set_ctrl(int fd, uint32_t ctrl)
42 {
43   if (ioctl(fd, NC_IOC_SET_CTRL, &ctrl) < 0) {
44     fprintf(stderr, "NC_IOC_SET_CTRL 0x%08x: %s\n",
45       ctrl, strerror(errno));
46     return -1;
47   }
48
49   return 0;
50 }
51
52 static int set_mu(int fd, uint32_t mu)
53 {
54   if (ioctl(fd, NC_IOC_SET_MU, &mu) < 0) {
55     fprintf(stderr, "NC_IOC_SET_MU 0x%04x: %s\n",
56       mu, strerror(errno));
57     return -1;
58   }
59
60   return 0;
61 }
62
63 static int pulse_reset(int fd)
64 {
65   struct nc_state state;
66   uint32_t ctrl;
67
68   if (nc_get_state(fd, &state) < 0)
69     return -1;
70
71   ctrl = state.ctrl | NC_CTRL_RESET_FILTER;
72   return set_ctrl(fd, ctrl);
73 }
74
75 static void print_samples(const struct nc_samples *samples)
76 {
77   printf("REF_SAMPLE=%+7d (0x%04x)  PRI_SAMPLE=%+7d (0x%04x)  "
78          "ERR_SAMPLE=%+7d (0x%04x)\n",
79          samples->ref, (uint16_t)samples->ref,
80          samples->primary, (uint16_t)samples->primary,
81          samples->error, (uint16_t)samples->error);
82 }
83
84 static int dump_samples(int fd)
85 {
86   struct nc_samples samples;
87
88   if (ioctl(fd, NC_IOC_READ_SAMPLES, &samples) < 0) {
89     fprintf(stderr, "NC_IOC_READ_SAMPLES: %s\n", strerror(errno));
90     return -1;
91   }
92
93   print_samples(&samples);
94   return 0;
95 }
96
97 static int dump_coeffs(int fd)
98 {
99   for (uint32_t i = 0; i < NC_TAP_COUNT; i++) {
100     struct nc_coeff coeff = { .index = i };
101
```

```
    if (ioctl(fd, NC_IOC_READ_COEFF, &coeff) < 0) {
      fprintf(stderr, "NC_IOC_READ_COEFF[%u]: %s\n",
        i, strerror(errno));
      return -1;
    }

    printf("w[%02u] = 0x%08x (%d)\n",
           i, (uint32_t)coeff.value, coeff.value);
  }

  return 0;
}

static int run_selftest(int fd)
{
  struct nc_coeff coeff = { .index = NC_TAP_COUNT - 1 };
  struct nc_samples samples;
  struct nc_state state;
  uint32_t value;
  int rc = 0;

  printf("Phase 6 register self-test\n");

  if (ioctl(fd, NC_IOC_GET_MAGIC, &value) < 0) {
    fprintf(stderr, "MAGIC read failed: %s\n", strerror(errno));
    return 1;
  }
  printf("MAGIC       read 0x%08x   %s\n",
         value, value == NC_MAGIC_VALUE ? "OK" : "FAIL");
  if (value != NC_MAGIC_VALUE)
    rc = 1;

  value = NC_CTRL_ENABLE | NC_CTRL_BYPASS;
  if (set_ctrl(fd, value) < 0 || nc_get_state(fd, &state) < 0)
    return 1;
  printf("CTRL        write 0x%08x  read 0x%08x   %s\n",
         value, state.ctrl,
         (state.ctrl & (NC_CTRL_ENABLE | NC_CTRL_BYPASS)) == value ?
         "OK" : "FAIL");
  if ((state.ctrl & (NC_CTRL_ENABLE | NC_CTRL_BYPASS)) != value)
    rc = 1;

  printf("STATUS      read 0x%08x   ", state.status);
  nc_print_status_bits(state.status);
  printf("   %s\n", (state.status & NC_STATUS_MCLK_ALIVE) ? "OK" : "FAIL");
  if (!(state.status & NC_STATUS_MCLK_ALIVE))
    rc = 1;
  if (!(state.status & NC_STATUS_CODEC_DATA_ALIVE))
    printf("WARN: codec_data_alive=0 until WM8731 is initialized and producing ADC data\n");

  value = NC_MU_DEFAULT;
  if (set_mu(fd, value) < 0 || nc_get_state(fd, &state) < 0)
    return 1;
  printf("MU          write 0x%08x  read 0x%08x   %s\n",
         value, state.mu, state.mu == value ? "OK" : "FAIL");
  if (state.mu != value)
    rc = 1;

  if (ioctl(fd, NC_IOC_READ_COEFF, &coeff) < 0) {
    fprintf(stderr, "COEFF_ADDR  write 0x%08x  read failed: %s\n",
      NC_TAP_COUNT - 1, strerror(errno));
    rc = 1;
  } else {
    printf("COEFF_ADDR  write 0x%08x  COEFF_DATA 0x%08x   OK\n",
           NC_TAP_COUNT - 1, (uint32_t)coeff.value);
  }

  if (ioctl(fd, NC_IOC_READ_SAMPLES, &samples) < 0) {
```

```
170     fprintf(stderr, "SAMPLES     read failed: %s\n", strerror(errno));
171     rc = 1;
172   } else {
173     printf("SAMPLES      ");
174     print_samples(&samples);
175   }
176 
177   value = NC_STATUS_W1C_MASK;
178   if (ioctl(fd, NC_IOC_CLEAR_STATUS, &value) < 0) {
179     fprintf(stderr, "STATUS clear failed: %s\n", strerror(errno));
180     rc = 1;
181   }
182 
183   value = NC_CTRL_BYPASS;
184   if (set_ctrl(fd, value) < 0)
185     rc = 1;
186   value = NC_MU_DEFAULT;
187   if (set_mu(fd, value) < 0)
188     rc = 1;
189 
190   return rc;
191 }
192 
193 static int interactive(int fd)
194 {
195   struct termios saved;
196   struct termios raw;
197   int have_termios = 0;
198   uint32_t ctrl = NC_CTRL_BYPASS;
199   uint32_t mu = NC_MU_DEFAULT;
200   time_t last_print = 0;
201 
202   if (tcgetattr(STDIN_FILENO, &saved) == 0) {
203     raw = saved;
204     raw.c_lflag &= ~(ICANON | ECHO);
205     raw.c_cc[VMIN] = 0;
206     raw.c_cc[VTIME] = 0;
207     if (tcsetattr(STDIN_FILENO, TCSANOW, &raw) == 0)
208       have_termios = 1;
209   }
210 
211   if (set_ctrl(fd, ctrl) < 0 || set_mu(fd, mu) < 0)
212     goto out_fail;
213 
214   printf("keys: b=bypass toggle, n=normal, r=reset, +/-=mu, q=quit\n");
215   while (!stop_requested) {
216     struct timeval tv = { .tv_sec = 0, .tv_usec = 100000 };
217     fd_set fds;
218     int sel;
219 
220     FD_ZERO(&fds);
221     FD_SET(STDIN_FILENO, &fds);
222     sel = select(STDIN_FILENO + 1, &fds, NULL, NULL, &tv);
223     if (sel > 0 && FD_ISSET(STDIN_FILENO, &fds)) {
224       char ch;
225 
226       if (read(STDIN_FILENO, &ch, 1) == 1) {
227         if (ch == 'q')
228           break;
229         if (ch == 'b')
230           ctrl ^= NC_CTRL_BYPASS;
231         else if (ch == 'n')
232           ctrl = NC_CTRL_ENABLE;
233         else if (ch == 'r')
234           (void)pulse_reset(fd);
235         else if (ch == '+') {
236           if (mu < NC_MU_MAX)
237             mu += 0x0100;
```

```
238          (void)set_mu(fd, mu);
239        } else if (ch == '-') {
240          if (mu > NC_MU_MIN)
241            mu -= 0x0100;
242          (void)set_mu(fd, mu);
243        }
244        (void)set_ctrl(fd, ctrl);
245      }
246    }
247
248    if (time(NULL) != last_print) {
249      struct nc_state state;
250
251      last_print = time(NULL);
252      if (nc_get_state(fd, &state) == 0) {
253        printf("\rCTRL=0x%08x MU=0x%04x ", state.ctrl,
254               state.mu);
255        nc_print_status_bits(state.status);
256        printf(" ref=%+7d pri=%+7d err=%+7d    ",
257               state.ref_sample, state.primary_sample,
258               state.error_sample);
259        fflush(stdout);
260      }
261    }
262  }
263
264  printf("\n");
265  if (have_termios)
266    tcsetattr(STDIN_FILENO, TCSANOW, &saved);
267  return 0;
268
269 out_fail:
270  if (have_termios)
271    tcsetattr(STDIN_FILENO, TCSANOW, &saved);
272  return 1;
273 }
274
275 int main(int argc, char **argv)
276 {
277   const char *device = NC_DEFAULT_DEVICE;
278   int do_selftest = 0;
279   int do_reset = 0;
280   int do_dump_samples = 0;
281   int do_dump_coeffs = 0;
282   int do_watch = 0;
283   int do_interactive = 0;
284   int ctrl_set = 0;
285   uint32_t ctrl_value = 0;
286   int mu_set = 0;
287   uint32_t mu_value = 0;
288   int fd;
289   int rc = 0;
290
291   for (int argi = 1; argi < argc; argi++) {
292     if (strcmp(argv[argi], "--selftest") == 0) {
293       do_selftest = 1;
294     } else if (strcmp(argv[argi], "--reset") == 0) {
295       do_reset = 1;
296     } else if (strcmp(argv[argi], "--enable") == 0) {
297       ctrl_value |= NC_CTRL_ENABLE;
298       ctrl_set = 1;
299     } else if (strcmp(argv[argi], "--bypass") == 0) {
300       ctrl_value |= NC_CTRL_BYPASS;
301       ctrl_set = 1;
302     } else if (strcmp(argv[argi], "--normal") == 0) {
303       ctrl_value = NC_CTRL_ENABLE;
304       ctrl_set = 1;
305     } else if (strcmp(argv[argi], "--mu") == 0 && argi + 1 < argc) {
```

```c
      if (nc_parse_u32(argv[++argi], &mu_value) < 0) {
        fprintf(stderr, "invalid mu value\n");
        return 2;
      }
      mu_set = 1;
    } else if (strcmp(argv[argi], "--dump-samples") == 0) {
      do_dump_samples = 1;
    } else if (strcmp(argv[argi], "--dump-coeffs") == 0) {
      do_dump_coeffs = 1;
    } else if (strcmp(argv[argi], "--watch") == 0) {
      do_watch = 1;
    } else if (strcmp(argv[argi], "--interactive") == 0) {
      do_interactive = 1;
    } else if (strcmp(argv[argi], "--device") == 0 && argi + 1 < argc) {
      device = argv[++argi];
    } else {
      usage(argv[0]);
      return 2;
    }
  }

  if (argc == 1)
    do_interactive = 1;

  signal(SIGINT, on_signal);
  signal(SIGTERM, on_signal);

  fd = nc_open_device(device);
  if (fd < 0)
    return 1;

  if (do_selftest)
    rc |= run_selftest(fd);
  if (do_reset)
    rc |= pulse_reset(fd) < 0;
  if (mu_set)
    rc |= set_mu(fd, mu_value) < 0;
  if (ctrl_set)
    rc |= set_ctrl(fd, ctrl_value) < 0;
  if (do_interactive)
    rc |= interactive(fd);

  do {
    if (do_dump_samples)
      rc |= dump_samples(fd) < 0;
    if (do_dump_coeffs)
      rc |= dump_coeffs(fd) < 0;
    if (do_watch)
      sleep_ms(250);
  } while (do_watch && !stop_requested);

  close(fd);
  return rc ? 1 : 0;
}
```

12.3.3 nc_monitor.c

```c
#include <math.h>
#include <signal.h>
#include <time.h>

#include "nc_user_common.h"

static volatile sig_atomic_t stop_requested;

static void on_signal(int signo)
{
  (void)signo;
```

```c
12   stop_requested = 1;
13 }
14
15 static void usage(const char *prog)
16 {
17   fprintf(stderr,
18     "usage: %s [--device /dev/noise_cancel] [--period-ms N]\n",
19     prog);
20 }
21
22 static double db_ratio(int32_t err, int32_t primary)
23 {
24   double e = fabs((double)err) + 1.0;
25   double d = fabs((double)primary) + 1.0;
26
27   return 20.0 * log10(e / d);
28 }
29
30 int main(int argc, char **argv)
31 {
32   const char *device = NC_DEFAULT_DEVICE;
33   uint32_t period_ms = 250;
34   struct timespec sleep_time;
35   int fd;
36
37   for (int argi = 1; argi < argc; argi++) {
38     if (strcmp(argv[argi], "--device") == 0 && argi + 1 < argc) {
39       device = argv[++argi];
40     } else if (strcmp(argv[argi], "--period-ms") == 0 &&
41         argi + 1 < argc) {
42       if (nc_parse_u32(argv[++argi], &period_ms) < 0 ||
43           period_ms == 0) {
44         fprintf(stderr, "invalid period\n");
45         return 2;
46       }
47     } else {
48       usage(argv[0]);
49       return 2;
50     }
51   }
52
53   fd = nc_open_device(device);
54   if (fd < 0)
55     return 1;
56
57   signal(SIGINT, on_signal);
58   signal(SIGTERM, on_signal);
59
60   sleep_time.tv_sec = period_ms / 1000;
61   sleep_time.tv_nsec = (long)(period_ms % 1000) * 1000000L;
62
63   printf("ctrl       mu      status                    ref      primary     error    |e|/|d|\n");
64   while (!stop_requested) {
65     struct nc_state state;
66
67     if (nc_get_state(fd, &state) < 0) {
68       close(fd);
69       return 1;
70     }
71
72     printf("0x%08x 0x%04x  ", state.ctrl, state.mu);
73     nc_print_status_bits(state.status);
74     printf("  %+8d %+8d %+8d  %+7.2f dB\n",
75            state.ref_sample, state.primary_sample, state.error_sample,
76            db_ratio(state.error_sample, state.primary_sample));
77     fflush(stdout);
78
```

```
    nanosleep(&sleep_time, NULL);
  }

  close(fd);
  return 0;
}
```

12.4 Verification

12.4.1 tb_lms.sv

```
`timescale 1ns/1ps

module tb_lms (
    input  logic                clk,
    input  logic                reset_n,
    input  logic                enable,
    input  logic                reset_filter,
    input  logic signed [15:0] d_in,
    input  logic signed [15:0] x_in,
    input  logic        [15:0] mu,
    input  logic                sample_valid,
    input  logic        [5:0]  coeff_addr,
    output logic signed [15:0] e_out,
    output logic signed [31:0] coeff_data,
    output logic                output_valid,
    output logic                saturation
);

    lms_filter dut (
        .clk(clk),
        .reset_n(reset_n),
        .enable(enable),
        .reset_filter(reset_filter),
        .d_in(d_in),
        .x_in(x_in),
        .mu(mu),
        .sample_valid(sample_valid),
        .coeff_addr(coeff_addr),
        .e_out(e_out),
        .coeff_data(coeff_data),
        .output_valid(output_valid),
        .saturation(saturation)
    );
endmodule
```

12.4.2 tb_lms.cpp

```
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <random>
#include <vector>

#include "Vtb_lms.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

static vluint64_t sim_time = 0;

double sc_time_stamp() {
    return static_cast<double>(sim_time);
}

static int16_t clamp_q15(double value) {
    if (value > 0.999969482421875) {
```

```
20          return 32767;
21      }
22      if (value < -1.0) {
23          return -32768;
24      }
25      return static_cast<int16_t>(std::lrint(value * 32768.0));
26  }
27
28  static double q15_to_double(int16_t value) {
29      return static_cast<double>(value) / 32768.0;
30  }
31
32  static void half_cycle(Vtb_lms *tb, VerilatedVcdC *trace = nullptr) {
33      tb->clk = !tb->clk;
34      tb->eval();
35      if (trace) {
36          trace->dump(sim_time);
37      }
38      sim_time++;
39  }
40
41  static void cycle(Vtb_lms *tb, VerilatedVcdC *trace = nullptr) {
42      half_cycle(tb, trace);
43      half_cycle(tb, trace);
44  }
45
46  static void reset_dut(Vtb_lms *tb, VerilatedVcdC *trace = nullptr) {
47      tb->clk = 0;
48      tb->reset_n = 0;
49      tb->enable = 0;
50      tb->reset_filter = 1;
51      tb->d_in = 0;
52      tb->x_in = 0;
53      tb->mu = 0x0200;
54      tb->sample_valid = 0;
55      tb->coeff_addr = 0;
56      tb->eval();
57
58      for (int i = 0; i < 4; i++) {
59          cycle(tb, trace);
60      }
61      tb->reset_n = 1;
62      cycle(tb, trace);
63      tb->reset_filter = 0;
64      cycle(tb, trace);
65  }
66
67  static bool wait_for_output(Vtb_lms *tb, int16_t *sample, VerilatedVcdC *trace = nullptr) {
68      for (int i = 0; i < 180; i++) {
69          cycle(tb, trace);
70          if (tb->output_valid) {
71              *sample = static_cast<int16_t>(tb->e_out);
72              return true;
73          }
74      }
75      return false;
76  }
77
78  static bool run_sample(Vtb_lms *tb, int16_t d, int16_t x, bool enable,
79                         int16_t *e, VerilatedVcdC *trace = nullptr) {
80      tb->enable = enable ? 1 : 0;
81      tb->d_in = d;
82      tb->x_in = x;
83      tb->sample_valid = 1;
84      cycle(tb, trace);
85      tb->sample_valid = 0;
86      return wait_for_output(tb, e, trace);
87  }
```

```
static int32_t read_coeff(Vtb_lms *tb, uint8_t addr) {
    tb->coeff_addr = addr & 0x3f;
    tb->eval();
    return static_cast<int32_t>(tb->coeff_data);
}

static bool check(bool condition, const char *name) {
    std::printf("%-42s %s\n", name, condition ? "OK" : "FAIL");
    return condition;
}

static bool reset_and_bypass_checks() {
    Vtb_lms tb;
    reset_dut(&tb);

    bool ok = true;
    ok &= check(tb.e_out == 0, "reset drives e_out to zero");
    ok &= check(tb.output_valid == 0, "reset leaves output_valid low");
    ok &= check(tb.saturation == 0, "reset clears saturation");
    ok &= check(read_coeff(&tb, 0) == 0, "reset clears coeff[0]");
    ok &= check(read_coeff(&tb, 63) == 0, "reset clears coeff[63]");

    int16_t e = 0;
    ok &= check(run_sample(&tb, 0x1234, 0x1234, false, &e), "disabled sample produces output");
    ok &= check(e == 0x1234, "disabled LMS passes d with zero coeffs");
    ok &= check(read_coeff(&tb, 0) == 0, "disabled LMS does not adapt coeff[0]");

    tb.reset_filter = 1;
    cycle(&tb);
    tb.reset_filter = 0;
    cycle(&tb);
    ok &= check(read_coeff(&tb, 0) == 0, "software reset clears coeff[0]");
    ok &= check(tb.saturation == 0, "software reset clears saturation");

    return ok;
}

static bool impulse_convergence_check() {
    Vtb_lms tb;
    reset_dut(&tb);
    tb.mu = 0x0200;

    std::mt19937 rng(0x48400001);
    std::uniform_real_distribution<double> dist(-0.45, 0.45);

    double tail_power = 0.0;
    int tail_count = 0;
    int16_t e = 0;
    bool ok = true;

    for (int n = 0; n < 12000; n++) {
        int16_t x = clamp_q15(dist(rng));
        ok &= run_sample(&tb, x, x, true, &e);
        if (!ok) {
            break;
        }
        if (n >= 10000) {
            double ed = q15_to_double(e);
            tail_power += ed * ed;
            tail_count++;
        }
    }

    const double rms_lsb = std::sqrt(tail_power / tail_count) * 32768.0;
    const int32_t w0 = read_coeff(&tb, 0);
    int64_t other_abs_sum = 0;
```

```
155     for (int i = 1; i < 8; i++) {
156         int32_t w = read_coeff(&tb, static_cast<uint8_t>(i));
157         other_abs_sum += (w < 0) ? -static_cast<int64_t>(w) : w;
158     }
159
160     std::printf("identity convergence: rms(e)=%.1f LSB w0=0x%08x other[1..7]_abs_sum=0x%llx\
    n",
161                 rms_lsb, static_cast<uint32_t>(w0),
162                 static_cast<unsigned long long>(other_abs_sum));
163     ok &= check(rms_lsb < 900.0, "identity residual falls below 900 LSB");
164     ok &= check(w0 > static_cast<int32_t>(0x30000000) &&
165                 w0 < static_cast<int32_t>(0x50000000),
166                 "coeff[0] converges near +1.0 Q2.30");
167     ok &= check(other_abs_sum < 0x20000000LL, "early nonzero taps stay small");
168     ok &= check(tb.saturation == 0, "identity convergence has no saturation");
169     return ok;
170 }
171
172 static bool saturation_guard_check() {
173     Vtb_lms tb;
174     reset_dut(&tb);
175     tb.mu = 0x7fff;
176
177     bool ok = true;
178     int16_t e = 0;
179     for (int n = 0; n < 3000 && !tb.saturation; n++) {
180         int16_t d = (n & 1) ? 32767 : -32768;
181         int16_t x = (n & 2) ? 32767 : -32768;
182         ok &= run_sample(&tb, d, x, true, &e);
183         if (!ok) {
184             break;
185         }
186     }
187
188     ok &= check(tb.saturation == 1, "oversized mu eventually sets saturation");
189     ok &= check(e <= 32767 && e >= -32768, "saturated output remains Q1.15 bounded");
190     return ok;
191 }
192
193 static bool run_snr_case(uint16_t mu, bool make_vcd,
194                          double *input_snr_db, double *output_snr_db) {
195     Vtb_lms *tb = new Vtb_lms;
196     VerilatedVcdC *trace = nullptr;
197
198     if (make_vcd) {
199         Verilated::traceEverOn(true);
200         trace = new VerilatedVcdC;
201         tb->trace(trace, 99);
202         trace->open("lms.vcd");
203     }
204
205     reset_dut(tb, trace);
206     tb->mu = mu;
207
208     const int samples = 20000;
209     const int taps = 64;
210     std::mt19937 rng(0x4840);
211     std::normal_distribution<double> noise_dist(0.0, 1.0);
212     std::vector<double> ref_hist(taps, 0.0);
213     std::vector<double> noise_path = {
214         0.62, -0.34, 0.21, 0.12, -0.08, 0.05, -0.03, 0.02
215     };
216
217     double primary_noise_power = 0.0;
218     double output_error_power = 0.0;
219     double signal_power = 0.0;
220     bool timed_out = false;
221
```

```
222     for (int n = 0; n < samples; n++) {
223         double white = 0.30 * noise_dist(rng);
224         for (int i = taps - 1; i > 0; i--) {
225             ref_hist[i] = ref_hist[i - 1];
226         }
227         ref_hist[0] = white;
228 
229         double correlated_noise = 0.0;
230         for (int i = 0; i < static_cast<int>(noise_path.size()); i++) {
231             correlated_noise += noise_path[i] * ref_hist[i];
232         }
233 
234         const double pi = 3.14159265358979323846;
235         double desired = 0.18 * std::sin(2.0 * pi * 1000.0 * n / 48000.0);
236         int16_t d_q15 = clamp_q15(desired + correlated_noise);
237         int16_t x_q15 = clamp_q15(ref_hist[0]);
238 
239         int16_t e_q15 = 0;
240         if (!run_sample(tb, d_q15, x_q15, true, &e_q15, trace)) {
241             timed_out = true;
242             break;
243         }
244 
245         if (n >= 1000) {
246             double d_actual = q15_to_double(d_q15);
247             double e_actual = q15_to_double(e_q15);
248             double s_actual = q15_to_double(clamp_q15(desired));
249             double input_noise = d_actual - s_actual;
250             double output_noise = e_actual - s_actual;
251             primary_noise_power += input_noise * input_noise;
252             output_error_power += output_noise * output_noise;
253             signal_power += s_actual * s_actual;
254         }
255     }
256 
257     if (trace) {
258         trace->close();
259         delete trace;
260     }
261 
262     bool saturated = tb->saturation;
263     delete tb;
264 
265     if (timed_out || saturated || output_error_power <= 0.0 ||
266         primary_noise_power <= 0.0 || signal_power <= 0.0) {
267         *input_snr_db = -999.0;
268         *output_snr_db = -999.0;
269         return false;
270     }
271 
272     *input_snr_db = 10.0 * std::log10(signal_power / primary_noise_power);
273     *output_snr_db = 10.0 * std::log10(signal_power / output_error_power);
274     return true;
275 }
276 
277 static bool snr_sweep_check() {
278     const uint16_t mu_values[] = {0x0010, 0x0020, 0x0040, 0x0080, 0x0100};
279     double best_input = 0.0;
280     double best_output = -999.0;
281     uint16_t best_mu = 0;
282 
283     for (uint16_t mu : mu_values) {
284         double input_snr = 0.0;
285         double output_snr = 0.0;
286         sim_time = 0;
287         bool ok = run_snr_case(mu, false, &input_snr, &output_snr);
288         if (ok) {
289             std::printf("mu=0x%04x input SNR: %.2f dB output SNR: %.2f dB improvement: %.2f
```

```
    dB\n",
                        mu, input_snr, output_snr, output_snr - input_snr);
            if (output_snr > best_output) {
                best_input = input_snr;
                best_output = output_snr;
                best_mu = mu;
            }
        } else {
            std::printf("mu=0x%04x failed: timeout, saturation, or invalid power estimate\n"
    , mu);
        }
    }

    if (best_mu != 0) {
        double input_snr = 0.0;
        double output_snr = 0.0;
        sim_time = 0;
        run_snr_case(best_mu, true, &input_snr, &output_snr);
    }

    std::printf("best mu: 0x%04x\n", best_mu);
    std::printf("input SNR: %.2f dB\n", best_input);
    std::printf("output SNR: %.2f dB\n", best_output);

    bool ok = true;
    ok &= check(best_mu != 0, "SNR sweep finds a stable mu");
    ok &= check(best_output >= best_input + 10.0, "SNR improves by at least 10 dB");
    return ok;
}

int main(int argc, char **argv) {
    Verilated::commandArgs(argc, argv);

    bool ok = true;
    sim_time = 0;
    ok &= reset_and_bypass_checks();
    sim_time = 0;
    ok &= impulse_convergence_check();
    sim_time = 0;
    ok &= saturation_guard_check();
    sim_time = 0;
    ok &= snr_sweep_check();

    std::printf("%s\n", ok ? "PASS" : "FAIL");
    return ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

12.4.3 tb_i2s.sv

```
`timescale 1ns/1ps

module tb_i2s (
    input  logic               clk,
    input  logic               reset_n,
    input  logic signed [15:0] dac_l_in,
    input  logic signed [15:0] dac_r_in,
    output logic               aud_bclk,
    output logic               aud_lrck,
    output logic               aud_dacdat,
    output logic signed [15:0] adc_l_out,
    output logic signed [15:0] adc_r_out,
    output logic               sample_valid
);

    logic aud_adcdat;

    assign aud_adcdat = aud_dacdat;

```

```
    i2s_dac_tx tx (
        .clk(clk),
        .reset_n(reset_n),
        .dac_l_in(dac_l_in),
        .dac_r_in(dac_r_in),
        .aud_bclk(aud_bclk),
        .aud_lrck(aud_lrck),
        .aud_dacdat(aud_dacdat)
    );

    i2s_adc_rx rx (
        .clk(clk),
        .reset_n(reset_n),
        .aud_bclk(aud_bclk),
        .aud_lrck(aud_lrck),
        .aud_adcdat(aud_adcdat),
        .adc_l_out(adc_l_out),
        .adc_r_out(adc_r_out),
        .sample_valid(sample_valid)
    );

endmodule
```

12.4.4 tb_i2s.cpp

```
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <vector>

#include "Vtb_i2s.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

static vluint64_t sim_time = 0;

double sc_time_stamp() {
    return static_cast<double>(sim_time);
}

struct SamplePair {
    int16_t left;
    int16_t right;
};

struct Monitor {
    int last_bclk = 0;
    int last_lrck = 0;
    int i2s_lrck = 0;
    int64_t last_bclk_edge = -1;
    int64_t last_lrck_edge = -1;
    int64_t last_valid = -1;
    int bclk_edges = 0;
    int lrck_edges = 0;
    int sample_valids = 0;
    bool tx_receiving = false;
    int tx_bit_count = 0;
    int tx_shift_lrck = 0;
    uint16_t tx_shift = 0;
    SamplePair tx_pair = {0, 0};
    bool tx_pair_valid = false;
    int tx_pairs = 0;
    bool ok = true;
};

static SamplePair pattern(int index) {
    uint32_t x = 0x9e3779b9u * static_cast<uint32_t>(index + 1);
    int16_t left = static_cast<int16_t>((x >> 4) ^ (x >> 19) ^ 0x1357u);
```

```
44     int16_t right = static_cast<int16_t>((x >> 11) ^ (x << 3) ^ 0x2468u);
45 
46     if (left == right) {
47         right = static_cast<int16_t>(right ^ 0x5a5a);
48     }
49     return {left, right};
50 }
51 
52 static bool check(bool condition, const char *name) {
53     std::printf("%-48s %s\n", name, condition ? "OK" : "FAIL");
54     return condition;
55 }
56 
57 static void monitor_posedge(Vtb_i2s *tb, Monitor *mon, int64_t cycle_count) {
58     bool bclk_rise = !mon->last_bclk && tb->aud_bclk;
59 
60     if (tb->aud_bclk != mon->last_bclk) {
61         if (mon->last_bclk_edge >= 0 &&
62             cycle_count - mon->last_bclk_edge != 2) {
63             std::printf("BCLK edge spacing error: got %lld cycles\n",
64                         static_cast<long long>(cycle_count - mon->last_bclk_edge));
65             mon->ok = false;
66         }
67         mon->last_bclk_edge = cycle_count;
68         mon->last_bclk = tb->aud_bclk;
69         mon->bclk_edges++;
70     }
71 
72     if (bclk_rise) {
73         bool i2s_lrck_changed = (tb->aud_lrck != mon->i2s_lrck);
74         mon->i2s_lrck = tb->aud_lrck;
75 
76         if (i2s_lrck_changed) {
77             mon->tx_receiving = true;
78             mon->tx_bit_count = 0;
79             mon->tx_shift_lrck = tb->aud_lrck;
80             mon->tx_shift = 0;
81         } else if (mon->tx_receiving) {
82             if (mon->tx_bit_count < 16) {
83                 uint16_t next_shift =
84                     static_cast<uint16_t>((mon->tx_shift << 1) |
85                                           (tb->aud_dacdat ? 1u : 0u));
86                 mon->tx_shift = next_shift;
87 
88                 if (mon->tx_bit_count == 15) {
89                     if (mon->tx_shift_lrck) {
90                         mon->tx_pair.right = static_cast<int16_t>(next_shift);
91                         mon->tx_pair_valid = true;
92                         mon->tx_pairs++;
93                     } else {
94                         mon->tx_pair.left = static_cast<int16_t>(next_shift);
95                     }
96                 }
97             }
98 
99             if (mon->tx_bit_count < 31) {
100                mon->tx_bit_count++;
101            }
102        }
103    }
104 
105    if (tb->aud_lrck != mon->last_lrck) {
106        if (mon->last_lrck_edge >= 0 &&
107            cycle_count - mon->last_lrck_edge != 128) {
108            std::printf("LRCK edge spacing error: got %lld cycles\n",
109                        static_cast<long long>(cycle_count - mon->last_lrck_edge));
110            mon->ok = false;
111        }
```

```
112         mon->last_lrck_edge = cycle_count;
113         mon->last_lrck = tb->aud_lrck;
114         mon->lrck_edges++;
115     }
116 
117     if (tb->sample_valid) {
118         if (!tb->aud_lrck) {
119             std::printf("sample_valid asserted outside right-channel slot\n");
120             mon->ok = false;
121         }
122         if (mon->last_valid >= 0 &&
123             cycle_count - mon->last_valid != 256) {
124             std::printf("sample_valid spacing error: got %lld cycles\n",
125                         static_cast<long long>(cycle_count - mon->last_valid));
126             mon->ok = false;
127         }
128         mon->last_valid = cycle_count;
129         mon->sample_valids++;
130     }
131 }
132 
133 static void half_cycle(Vtb_i2s *tb, VerilatedVcdC *trace) {
134     tb->clk = !tb->clk;
135     tb->eval();
136     if (trace) {
137         trace->dump(sim_time);
138     }
139     sim_time++;
140 }
141 
142 static void cycle(Vtb_i2s *tb, VerilatedVcdC *trace, Monitor *mon,
143                   int64_t *cycle_count) {
144     half_cycle(tb, trace);
145     if (tb->clk) {
146         (*cycle_count)++;
147         monitor_posedge(tb, mon, *cycle_count);
148     }
149     half_cycle(tb, trace);
150 }
151 
152 static bool wait_for_valid(Vtb_i2s *tb, VerilatedVcdC *trace, Monitor *mon,
153                            int64_t *cycle_count, SamplePair *observed) {
154     for (int i = 0; i < 400; i++) {
155         cycle(tb, trace, mon, cycle_count);
156         if (tb->sample_valid) {
157             observed->left = static_cast<int16_t>(tb->adc_l_out);
158             observed->right = static_cast<int16_t>(tb->adc_r_out);
159             if (!mon->tx_pair_valid) {
160                 std::printf("independent TX decoder had no pair at sample_valid\n");
161                 mon->ok = false;
162             } else if (observed->left != mon->tx_pair.left ||
163                        observed->right != mon->tx_pair.right) {
164                 std::printf("RX differs from independent TX decode: rx L=0x%04x R=0x%04x tx
    L=0x%04x R=0x%04x\n",
165                             static_cast<uint16_t>(observed->left),
166                             static_cast<uint16_t>(observed->right),
167                             static_cast<uint16_t>(mon->tx_pair.left),
168                             static_cast<uint16_t>(mon->tx_pair.right));
169                 mon->ok = false;
170             }
171             mon->tx_pair_valid = false;
172             return true;
173         }
174     }
175     return false;
176 }
177 
178 int main(int argc, char **argv) {
```

```cpp
    Verilated::commandArgs(argc, argv);
    Verilated::traceEverOn(true);

    Vtb_i2s tb;
    VerilatedVcdC trace;
    tb.trace(&trace, 99);
    trace.open("i2s_loopback.vcd");

    Monitor mon;
    int64_t cycle_count = 0;
    bool ok = true;

    tb.clk = 0;
    tb.reset_n = 0;
    tb.dac_l_in = 0;
    tb.dac_r_in = 0;
    tb.eval();

    for (int i = 0; i < 12; i++) {
        cycle(&tb, &trace, &mon, &cycle_count);
    }

    tb.reset_n = 1;
    SamplePair first = pattern(0);
    tb.dac_l_in = first.left;
    tb.dac_r_in = first.right;

    std::printf("Testing 1024 sample pairs...\n");

    SamplePair observed = {0, 0};
    bool synced = false;
    for (int warmup = 0; warmup < 8; warmup++) {
        ok &= wait_for_valid(&tb, &trace, &mon, &cycle_count, &observed);
        if (!ok) {
            break;
        }
        if (observed.left == first.left && observed.right == first.right) {
            synced = true;
            break;
        }
    }

    ok &= check(synced, "loopback synchronizes on first pattern");

    int matched = 0;
    if (ok) {
        for (int i = 0; i < 1024; i++) {
            SamplePair expected = pattern(i);

            if (i != 0) {
                ok &= wait_for_valid(&tb, &trace, &mon, &cycle_count, &observed);
            }
            if (!ok) {
                break;
            }

            if (observed.left != expected.left || observed.right != expected.right) {
                std::printf("mismatch[%d]: got L=0x%04x R=0x%04x expected L=0x%04x R=0x%04x\n",
                            i,
                            static_cast<uint16_t>(observed.left),
                            static_cast<uint16_t>(observed.right),
                            static_cast<uint16_t>(expected.left),
                            static_cast<uint16_t>(expected.right));
                ok = false;
                break;
            }

```

```
            matched++;
            if (i + 1 < 1024) {
                SamplePair next = pattern(i + 1);
                tb.dac_l_in = next.left;
                tb.dac_r_in = next.right;
            }
        }
    }

    ok &= check(matched == 1024, "1024/1024 samples matched bit-exactly");
    ok &= check(mon.ok, "clock and sample_valid cadence stayed exact");
    ok &= check(mon.bclk_edges > 1024 * 100, "BCLK toggled continuously");
    ok &= check(mon.lrck_edges >= 1024 * 2, "LRCK toggled once per channel slot");
    ok &= check(mon.sample_valids >= 1024, "sample_valid pulsed once per stereo frame");
    ok &= check(mon.tx_pairs >= 1024, "independent TX decoder saw each stereo frame");

    trace.close();

    std::printf("%s: %d/1024 samples matched bit-exactly\n",
                ok ? "PASS" : "FAIL", matched);
    return ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```