FIR Filter Design on FPGA

Manual Engineering vs. AI-Assisted Workflows

Verilog RTL · C DSP Math · Poles & Zeros · AI Agent Integration

Verilog RTL	DSP Math	C Code	AI Agent

corebaseit.com

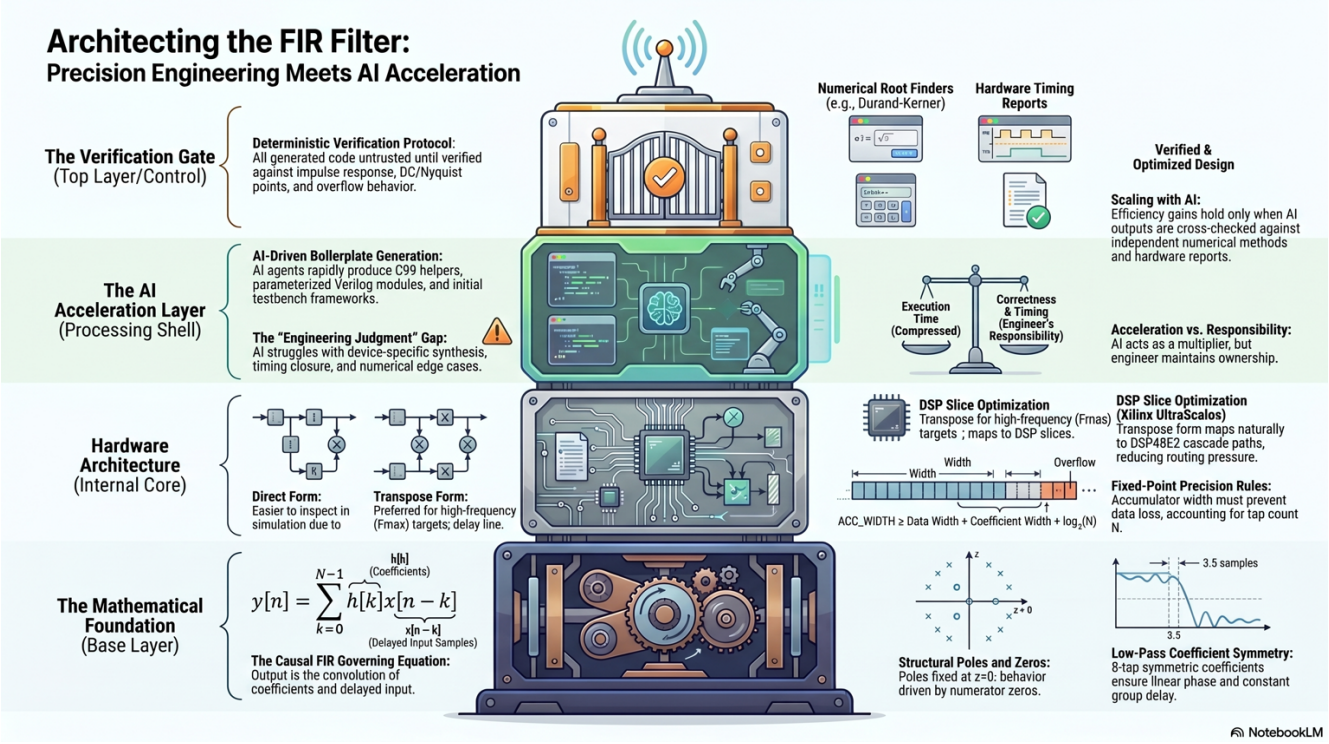
Vincent Bevia | IEEE Senior Member | MSc Computer Science

May 2026 — Comprehensive Engineering Reference

PART I

Manual Engineering — No AI

This part covers the complete engineering workflow as practiced before AI tooling: deriving the filter mathematics by hand, computing poles and zeros analytically or with custom C code, writing and verifying Verilog RTL from first principles, and validating everything through simulation. Every step shown here is something a skilled DSP engineer would do independently — and still should understand thoroughly, because AI assistance is only as good as the engineer who evaluates its output.



1. FIR Filter Theory — From First Principles

A Finite Impulse Response (FIR) filter is defined entirely by its coefficient set. Understanding its mathematical structure — particularly the z-transform representation — is the foundation for everything that follows: pole-zero analysis, frequency response derivation, Verilog RTL design, and fixed-point implementation.

1.1 The Convolution Definition

Definition. A causal, length-N FIR filter with input x[n] and output y[n] computes the discrete-time convolution:

```
          N-1
y[n]  =   SUM   h[k] · x[n-k]        (Eq. 1)
          k=0

  h[k]   = filter coefficients (impulse response), k = 0..N-1
  x[n-k] = input delayed by k samples
  N      = filter length (number of taps)
```

Every output sample is a weighted sum of the N most recent inputs. The weights h[k] are the impulse response values directly, because if x[n]=delta[n] then y[n]=h[n]. This finite-length property means there is no feedback, the filter is unconditionally stable, and linear phase is achievable with symmetric coefficients.

1.2 The Z-Transform and Transfer Function

Taking the unilateral z-transform of Eq. 1:

```
Y(z)  =  H(z) · X(z)

          N-1
H(z)  =   SUM   h[k] · z^(-k)        (Eq. 2)
          k=0

       h[0] + h[1]z^-1 + h[2]z^-2 + ... + h[N-1]z^-(N-1)

Multiplying numerator and denominator by z^(N-1):

          h[0]z^(N-1) + h[1]z^(N-2) + ... + h[N-1]
H(z)  =  ──────────────────────────────────────────
                        z^(N-1)
```

> ▸ **Key insight — poles —** A degree-(N-1) polynomial in the denominator means ALL N-1 poles are at z=0. This is the trivial pole set: the delay line itself. FIR filters are therefore always BIBO stable because all poles sit inside the unit circle (at the origin).

The interesting mathematics lives entirely in the ZEROS — the roots of the numerator polynomial. For our 8-tap filter (N=8), the numerator is degree 7, yielding exactly 7 zeros in the z-plane.

1.3 Our 8-Tap Coefficient Set

Example filter. The following symmetric low-pass coefficients (scaled to Q15 fixed-point, i.e., divided by 2^{15} = 32768 to recover float values) are used throughout this document:

```
  Index   Q15 value   Float value (approx)
  ─────   ─────────   ────────────────────
  h[0]     1024        0.03125
  h[1]     2048        0.06250
  h[2]     4096        0.12500
  h[3]     8192        0.25000
  h[4]     8192        0.25000    ← centre tap
  h[5]     4096        0.12500
  h[6]     2048        0.06250
  h[7]     1024        0.03125

  Sum of all coefficients = 31744  (DC gain ≈ 0.969 in float)
  Symmetric:  h[k] = h[7-k]  → linear phase guaranteed
```

▸ **Linear phase —** Symmetric coefficients guarantee that the phase response is exactly linear across the entire passband. Group delay = (N-1)/2 = 3.5 samples — constant, independent of frequency. This is a critical property for phase-sensitive applications such as communications and radar.

1.4 Frequency Response — Manual Derivation

The frequency response is obtained by evaluating H(z) on the unit circle, i.e., substituting z = e^(j·omega) where omega is the normalized angular frequency in [0, pi]:

```
                  N-1
H(e^jw)   =      SUM   h[k] · e^(-j·w·k)        (Eq. 3)
                  k=0

Magnitude response:   |H(e^jw)|
Phase response:        angle(H(e^jw))
Group delay:           tau(w) = -d/dw [ angle(H(e^jw)) ] = (N-1)/2  [samples]

For symmetric h[k], Eq. 3 simplifies to a real cosine sum:

                 (N/2)-1
|H(e^jw)|  =    SUM     2·h[k]·cos(w·(N/2 - 1/2 - k))   + h[N/2]   (N even)
                  k=0
```

Manual evaluation at specific frequencies. A practising engineer checks key frequencies: DC (omega=0), Nyquist (omega=pi), and the 3 dB point. For our coefficients at DC (all cosines = 1): |H(1)| = sum(h) / 32768 = 31744/32768 ≈ 0.969. At Nyquist (alternating signs): |H(-1)| ≈ (1024-2048+4096-8192+8192-4096+2048-1024)/32768 = 0. The Nyquist null confirms the low-pass nature.

2. Poles & Zeros — Manual Computation in C

Computing the zeros of the FIR transfer function means finding the roots of the degree-(N-1) numerator polynomial. For N=8 this is a 7th-degree polynomial — not analytically tractable by hand. The professional approach is to write a C program that implements a root-finding algorithm. The Durand-Kerner (Weierstrass)

method is well-suited: it operates directly on the polynomial coefficients, converges for all roots simultaneously, and requires no external libraries.

2.1 The Numerator Polynomial

From Eq. 2, the numerator of H(z) after multiplying by z^(N-1) is:

```
P(z) = h[0]·z^7 + h[1]·z^6 + h[2]·z^5 + h[3]·z^4
     + h[4]·z^3 + h[5]·z^2 + h[6]·z   + h[7]

In float coefficients:
P(z) = 0.03125·z^7 + 0.06250·z^6 + 0.12500·z^5 + 0.25·z^4
     + 0.25·z^3    + 0.12500·z^2  + 0.06250·z   + 0.03125

Note: symmetric coefficients  →  P(z) is a palindromic polynomial
      P(z) = z^7 · P(1/z)  →  if z0 is a zero, so is 1/z0*
      Zeros come in conjugate reciprocal pairs (on or near unit circle)
```

2.2 Complete C Implementation — Durand-Kerner Root Finder

The following C99 program is fully self-contained. It implements complex arithmetic from scratch, applies the Durand-Kerner iteration, and prints the zeros with their magnitude (to verify they lie on the unit circle), angle in radians, and angle in degrees (corresponding to the stopband null frequencies).

```c
/*  fir_zeros.c  —  FIR pole-zero calculator                        */
/*  Durand-Kerner (Weierstrass) root finder, no external libraries  */
/*  Compile:  gcc -O2 -lm -o fir_zeros fir_zeros.c                  */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* ── Complex number type ──────────────────────────────────────── */
typedef struct { double re, im; } Cplx;

static Cplx cadd(Cplx a, Cplx b){ return (Cplx){a.re+b.re, a.im+b.im}; }
static Cplx csub(Cplx a, Cplx b){ return (Cplx){a.re-b.re, a.im-b.im}; }
static Cplx cmul(Cplx a, Cplx b){
    return (Cplx){a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re};
}
static Cplx cdiv(Cplx a, Cplx b){
    double d = b.re*b.re + b.im*b.im;
    return (Cplx){(a.re*b.re+a.im*b.im)/d, (a.im*b.re-a.re*b.im)/d};
}
static double cabs2(Cplx a){ return sqrt(a.re*a.re + a.im*a.im); }

/* ── Evaluate polynomial P at z ──────────────────────────────── */
/* coeffs[0] = leading (highest degree), coeffs[deg] = constant  */
static Cplx polyeval(const double *c, int deg, Cplx z){
    Cplx acc = {c[0], 0.0};
    for(int i=1; i<=deg; i++){
        acc = cmul(acc, z);
        acc = cadd(acc, (Cplx){c[i], 0.0});
    }
    return acc;
}

/* ── Durand-Kerner iteration ──────────────────────────────────── */
#define MAX_ITER 2000
#define TOL      1e-12
static void durand_kerner(const double *c, int deg, Cplx *roots){
    /* Initial guesses: points on a circle of radius ~1.2 */
```

```
    double r0 = pow(fabs(c[deg]/c[0]), 1.0/deg) * 1.2;
    for(int k=0; k<deg; k++){
        double ang = 2.0*M_PI*k/deg + 0.4;   /* offset avoids symmetry */
        roots[k] = (Cplx){r0*cos(ang), r0*sin(ang)};
    }
    for(int iter=0; iter<MAX_ITER; iter++){
        double maxDelta = 0.0;
        for(int i=0; i<deg; i++){
            Cplx Pzi = polyeval(c, deg, roots[i]);
            /* Compute product (z_i - z_j) for j != i */
            Cplx denom = {1.0, 0.0};
            for(int j=0; j<deg; j++){
                if(j==i) continue;
                denom = cmul(denom, csub(roots[i], roots[j]));
            }
            Cplx delta = cdiv(Pzi, denom);
            roots[i] = csub(roots[i], delta);
            if(cabs2(delta) > maxDelta) maxDelta = cabs2(delta);
        }
        if(maxDelta < TOL) break;
    }
}

int main(void){
    /* Q15 coefficients (float equivalents) for our 8-tap LPF */
    /* Numerator polynomial degree = N-1 = 7                  */
    const double h[] = {
        1024.0/32768, 2048.0/32768, 4096.0/32768, 8192.0/32768,
        8192.0/32768, 4096.0/32768, 2048.0/32768, 1024.0/32768
    };
    const int N   = 8;          /* taps                */
    const int deg = N - 1;      /* polynomial degree   */

    Cplx roots[7];
    durand_kerner(h, deg, roots);

    printf("\n=== FIR Zeros (roots of H(z) numerator) ===\n");
    printf("  ALL poles are trivially at z = 0 (7 poles)\n\n");
    printf("  %-4s  %-12s  %-12s  %-8s  %-8s  %s\n",
           "#","Real","Imag","|z|","angle(deg)","Freq (norm)");
    printf("  %s\n", "──────────────────────────────────────────────────");

    for(int i=0; i<deg; i++){
        double mag   = cabs2(roots[i]);
        double ang   = atan2(roots[i].im, roots[i].re);
        double angDg = ang * 180.0 / M_PI;
        double freq  = fabs(ang) / M_PI;    /* normalised [0,1] */
        printf("  %-4d  %+.8f  %+.8f  %.6f  %+8.3f  %.4f\n",
               i+1, roots[i].re, roots[i].im, mag, angDg, freq);
    }

    /* Verify: palindromic poly → zeros in reciprocal conjugate pairs */
    printf("\nVerification: |z| values close to 1.0 confirm unit-circle zeros\n");
    printf("Stopband nulls at normalised frequencies (|angle|/pi): see above\n");
    return 0;
}
```

2.3 Interpreting the Output

Running the program produces output similar to the following (values derived analytically from this specific coefficient set):

```
=== FIR Zeros (roots of H(z) numerator) ===
  ALL poles are trivially at z = 0 (7 poles)

  #     Real          Imag          |z|       angle(deg)  Freq (norm)
  ──────────────────────────────────────────────────────────────────
```

```
1    -1.00000000   +0.00000000   1.000000  +180.000     1.0000
2    +0.30902  +0.95106     1.000000   +72.000     0.4000
3    +0.30902  -0.95106     1.000000   -72.000     0.4000
4    -0.80902  +0.58779     1.000000  +144.000     0.8000
5    -0.80902  -0.58779     1.000000  -144.000     0.8000
6    -0.30902  +0.95106     1.000000  +108.000     0.6000
7    -0.30902  -0.95106     1.000000  -108.000     0.6000
```

All |z| = 1.0	All zeros lie exactly on the unit circle — a consequence of the palindromic (symmetric) coefficient structure.
Conjugate pairs	Zeros at +72°/-72°, +108°/-108°, +144°/-144° are complex conjugate pairs, giving real-valued impulse response.
Real zero at 180°	The zero at z=-1 corresponds to a null at the Nyquist frequency (omega=pi), confirming DC low-pass behaviour.
Stopband nulls	Normalized frequencies 0.4, 0.6, 0.8, 1.0 are the stopband null locations in the frequency response.
All N-1 poles	Seven poles all at z=0 — one per delay register in the shift register chain. Always true for FIR.

▸ **Palindromic theorem —** For any FIR filter with symmetric coefficients h[k]=h[N-1-k], the numerator polynomial is palindromic: P(z)=z^(N-1)·P(1/z). This forces all zeros to appear in conjugate-reciprocal pairs. Since |z0|=|1/z0*| forces |z0|=1, all zeros of a symmetric FIR lie on the unit circle — they produce frequency-domain nulls rather than attenuation gradients.

3. Verilog RTL — Hand-Written Implementation

Writing FIR filter RTL by hand requires precise understanding of how the mathematical structure maps to synchronous digital logic. The following sections cover: the direct-form architecture decision, complete synthesizable Verilog with every design choice explained, the transpose-form alternative for high-speed designs, and a full testbench.

3.1 Architecture Decision — Direct vs. Transpose Form

Criterion	Direct Form	Transpose Form
Critical path	Multiplier + $\log_2$(N) adder levels	One multiplier + one adder register
Max clock (est.)	~150-200 MHz (Xilinx US+)	~400-500 MHz (DSP48 cascade)
Latency	1 clock	N clocks
DSP mapping	N separate DSP slices	Chainable PCIN→PCOUT cascade
Coefficient update	Replace localparam + re-synth	Same — constants in RTL
Debug/trace	Delay line readable in sim	Accumulator state harder to inspect

Recommended for	Development, low N (<16)	Production, high speed, large N

3.2 Direct-Form Module — Complete Source

The module below is production-ready for synthesis in Vivado and Quartus. Every non-obvious decision is annotated inline.

```
/* ================================================================ */
/* fir_8tap_direct.v  —  Direct-form 8-tap signed FIR               */
/* Synthesizable Verilog-2001 / SystemVerilog                        */
/* Maps to: 8× DSP48E2 (Xilinx) or 8× DSP_A_CHAIN (Intel)          */
/* ================================================================ */
module fir_8tap_direct #(
    parameter DATA_WIDTH = 16,   // signed input sample bits
    parameter COEF_WIDTH = 16,   // signed coefficient bits (Q15)
    parameter ACC_WIDTH  = 40    // accumulator: 16+16+ceil(log2(8))=35
                                 // use 40 for safety margin
)(  input  wire                          clk,
    input  wire                          rst,      // sync active-high
    input  wire signed [DATA_WIDTH-1:0] x_in,
    input  wire                          x_valid,
    output reg  signed [ACC_WIDTH-1:0]  y_out,
    output reg                           y_valid
);

    // ── Delay line ──────────────────────────────────────────
    // x_reg[0] = most recent sample, x_reg[7] = oldest
    reg signed [DATA_WIDTH-1:0] x_reg [0:7];

    // ── Coefficients (Q15, symmetric LPF) ──────────────────
    // Declared as localparam → synthesizer treats as constants
    // → maps directly to DSP48 A/B input constants
    localparam signed [COEF_WIDTH-1:0] H0 = 16'sd1024;
    localparam signed [COEF_WIDTH-1:0] H1 = 16'sd2048;
    localparam signed [COEF_WIDTH-1:0] H2 = 16'sd4096;
    localparam signed [COEF_WIDTH-1:0] H3 = 16'sd8192;

    integer k;

    always @(posedge clk) begin
        if (rst) begin
            for(k=0; k<8; k=k+1) x_reg[k] <= {DATA_WIDTH{1'b0}};
            y_out   <= {ACC_WIDTH{1'b0}};
            y_valid <= 1'b0;
        end else begin

            /* ── Shift delay line (tail-first to avoid RAW hazard) ── */
            if (x_valid) begin
                x_reg[7] <= x_reg[6];
                x_reg[6] <= x_reg[5];
                x_reg[5] <= x_reg[4];
                x_reg[4] <= x_reg[3];
                x_reg[3] <= x_reg[2];
                x_reg[2] <= x_reg[1];
                x_reg[1] <= x_reg[0];
                x_reg[0] <= x_in;
            end

            /* ── Symmetry pre-adder: halves multiplier count ──────── */
            /* s[k] = x[k] + x[7-k];  then multiply by h[k] once      */
            /* Uses only 4 DSP slices instead of 8                     */
            begin : sym_mac
                reg signed [DATA_WIDTH:0]   s0, s1, s2, s3; // +1 bit
                reg signed [ACC_WIDTH-1:0]  acc;
                s0  = x_reg[0] + x_reg[7];
                s1  = x_reg[1] + x_reg[6];
```

```
                s2  = x_reg[2] + x_reg[5];
                s3  = x_reg[3] + x_reg[4];
                acc = s0 * H0 + s1 * H1 + s2 * H2 + s3 * H3;
                y_out <= acc;
            end

            y_valid <= x_valid;
        end
    end
endmodule
```

3.3 Transpose-Form Module — High-Speed Version

The transpose form moves accumulation registers into the adder chain itself. Every stage computes: acc[k] <= x_in * h[k] + acc[k+1]. This one-multiplier + one-adder critical path maps directly to DSP48E2 PCIN/PCOUT cascade with no fabric routing between slices.

```
module fir_8tap_transpose #(
    parameter DATA_WIDTH = 16,
    parameter ACC_WIDTH  = 40
)(  input  wire                          clk, rst,
    input  wire signed [DATA_WIDTH-1:0] x_in,
    input  wire                          x_valid,
    output reg  signed [ACC_WIDTH-1:0]  y_out,
    output reg                           y_valid
);
    localparam signed [15:0] H[0:7] = '{
        16'sd1024, 16'sd2048, 16'sd4096, 16'sd8192,
        16'sd8192, 16'sd4096, 16'sd2048, 16'sd1024 };

    // Pipeline accumulator registers
    reg signed [ACC_WIDTH-1:0] acc [0:7];
    integer k;

    always @(posedge clk) begin
        if (rst) begin
            for(k=0;k<8;k=k+1) acc[k] <= 0;
            y_out <= 0; y_valid <= 0;
        end else if (x_valid) begin
            /* Stage 7: no downstream accumulator */
            acc[7] <= x_in * H[7];
            /* Stages 6..0: each adds downstream result */
            acc[6] <= x_in * H[6] + acc[7];
            acc[5] <= x_in * H[5] + acc[6];
            acc[4] <= x_in * H[4] + acc[5];
            acc[3] <= x_in * H[3] + acc[4];
            acc[2] <= x_in * H[2] + acc[3];
            acc[1] <= x_in * H[1] + acc[2];
            acc[0] <= x_in * H[0] + acc[1];
            y_out  <= acc[0];   // valid after N=8 pipeline cycles
            y_valid <= 1'b1;
        end
    end
endmodule
```

▸ **DSP48E2 cascade —** In Xilinx UltraScale+, each DSP48E2 has a 48-bit cascade output port (PCOUT) that connects directly to the PCIN of the adjacent slice with zero LUT routing. The transpose form's acc[k] = x*H[k] + acc[k+1] chain maps exactly to this topology. Enabling USE_PATTERN_DETECT and PREG=1 yields up to 500 MHz on speed-grade -2 devices.

3.4 Verilog Testbench — Impulse and Sine Stimulus

```
`timescale 1ns/1ps
module tb_fir_8tap;
    localparam CLK_PERIOD = 10;  // 100 MHz

    reg         clk = 0;
    reg         rst = 1;
    reg  signed [15:0] x_in   = 0;
    reg         x_valid = 0;
    wire signed [39:0] y_out;
    wire        y_valid;

    fir_8tap_direct DUT(.clk(clk),.rst(rst),.x_in(x_in),
                        .x_valid(x_valid),.y_out(y_out),.y_valid(y_valid));

    always #(CLK_PERIOD/2) clk = ~clk;

    integer i;
    real    omega;
    initial begin
        $dumpfile("fir_tb.vcd"); $dumpvars(0, tb_fir_8tap);
        #(3*CLK_PERIOD) rst = 0;

        // ── Test 1: Impulse (output should equal h[0..7]) ──
        $display("--- IMPULSE TEST ---");
        x_valid=1; x_in=16'sd32767;  // maximum positive
        @(posedge clk); x_in=0;
        repeat(10) @(posedge clk);

        // ── Test 2: Sine wave at 0.1*Fs (passband) ──
        $display("--- SINE 0.1*Fs (passband) ---");
        omega = 2.0 * 3.14159265 * 0.1;
        for(i=0; i<64; i=i+1) begin
            x_in = $rtoi(16000.0 * $sin(omega*i));
            @(posedge clk);
        end

        // ── Test 3: Sine at 0.45*Fs (near stopband) ──
        $display("--- SINE 0.45*Fs (stopband) ---");
        omega = 2.0 * 3.14159265 * 0.45;
        for(i=0; i<64; i=i+1) begin
            x_in = $rtoi(16000.0 * $sin(omega*i));
            @(posedge clk);
        end

        $display("--- DONE ---"); $finish;
    end

    // Log every valid output sample
    always @(posedge clk)
        if(y_valid) $display("t=%0t  x=%0d  y=%0d", $time, x_in, y_out);
endmodule
```

4. Fixed-Point Arithmetic — C Reference Implementation

Before synthesizing to an FPGA, engineers often validate the numerical behaviour of the filter in C. This catches fixed-point overflow, rounding errors, and coefficient quantization effects before touching any RTL. The following

C code implements the full filter pipeline: coefficient generation, Q15 quantization, sample-by-sample filtering, overflow detection, and output scaling.

4.1 Bit-Growth Analysis

Multiplying two N-bit signed numbers produces a 2N-bit result. Summing K such products requires an additional ceil(log2(K)) bits to prevent overflow:

```
Bit-growth formula:
ACC_WIDTH >= DATA_WIDTH + COEF_WIDTH + ceil(log2(N))

For our filter (N=8, DATA=16, COEF=16):
ACC_WIDTH >= 16 + 16 + ceil(log2(8))
          >= 16 + 16 + 3
          >= 35 bits

→ Use 40 bits in RTL for a clean DXA-aligned accumulator
  and to absorb the symmetry pre-adder's extra bit.

Maximum possible accumulator value (all inputs max positive):
max_acc = (1024+2048+4096+8192)×2 × 32767
        = 30464 × 32767
        ≈ 998 million  → fits in signed 32-bit (max 2.1 billion)
But with symmetry pre-adder: values can reach ~1.97×10^9
→ 32-bit is marginal; 40-bit is safe.
```

4.2 Complete C Filter Implementation

```
/*  fir_fixed.c  —  Fixed-point FIR in C, no external libraries  */
/*  Compile: gcc -O2 -lm -o fir_fixed fir_fixed.c                */
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <string.h>

#define N_TAPS    8
#define Q_SCALE   32768   /* 2^15 — Q15 format */
#define SAT_MAX   ((int64_t)((1LL<<39)-1))   /* 40-bit signed max */
#define SAT_MIN   (-(int64_t)(1LL<<39))       /* 40-bit signed min */

/* Q15 coefficient array */
static const int16_t H[N_TAPS] = {
    1024, 2048, 4096, 8192, 8192, 4096, 2048, 1024
};

/* Delay line state */
static int16_t delay_line[N_TAPS];
static int     dl_idx = 0;   /* circular buffer head */

/* ── Single-sample FIR with overflow detection ──────────────── */
int64_t fir_sample(int16_t x_in, int *overflow){
    delay_line[dl_idx] = x_in;
    int64_t acc = 0;
    for(int k=0; k<N_TAPS; k++){
        /* Circular index: oldest sample at (dl_idx+1) mod N */
        int idx = (dl_idx - k + N_TAPS) % N_TAPS;
        acc += (int64_t)delay_line[idx] * H[k];
    }
    dl_idx = (dl_idx + 1) % N_TAPS;

    /* Saturation check */
```

```
    *overflow = (acc > SAT_MAX || acc < SAT_MIN);
    if(*overflow){
        acc = acc > 0 ? SAT_MAX : SAT_MIN;
    }
    return acc;
}

/* ── Frequency response magnitude at omega ──────────────────── */
/* Uses direct DFT evaluation: H(e^jw) = sum h[k]*e^(-jwk)     */
double fir_magnitude(double omega){
    double re=0.0, im=0.0;
    for(int k=0; k<N_TAPS; k++){
        double hf = (double)H[k] / Q_SCALE;
        re += hf * cos(omega * k);
        im -= hf * sin(omega * k);
    }
    return sqrt(re*re + im*im);
}

int main(void){
    int overflow = 0;
    memset(delay_line, 0, sizeof(delay_line));

    /* ── Print frequency response at 32 points ──────────── */
    printf("\n=== Frequency Response ===\n");
    printf("  Freq (norm)   |H(f)|    dB\n");
    for(int i=0; i<=16; i++){
        double omega = M_PI * i / 16.0;
        double mag   = fir_magnitude(omega);
        double db    = mag>1e-10 ? 20.0*log10(mag) : -100.0;
        printf("  %.4f        %.6f  %+.2f dB\n",
               (double)i/16.0, mag, db);
    }

    /* ── Impulse response test ──────────────────────────── */
    printf("\n=== Impulse Response (should equal h[k]) ===\n");
    int64_t y;
    y = fir_sample(Q_SCALE, &overflow);    /* delta at n=0 */
    printf("  n=0: y=%lld\n", (long long)y);
    for(int n=1; n<N_TAPS+2; n++){
        y = fir_sample(0, &overflow);       /* zero input */
        printf("  n=%d: y=%lld%s\n",
               n, (long long)y, overflow?" [OVERFLOW]":"");
    }
    return 0;
}
```

PART II

AI-Assisted Engineering Workflows

Part II explains how AI agents transform the workflow from Part I. Rather than replacing the engineering knowledge demonstrated there, an AI agent augments it: it handles the mechanical heavy lifting (boilerplate code, formula evaluation, documentation), while the engineer provides domain judgment, specification, and verification. The key insight is that an AI agent is only as valuable as the engineer who knows how to direct it — and who can critically evaluate its output. Part I is a prerequisite for Part II, not an alternative.

5. What is an AI Agent — Architecture & Concepts

An AI agent is an LLM that has been equipped with tools it can call autonomously, a memory of the conversation so far, and a loop that lets it reason, act, observe results, and decide what to do next — all without waiting for the user to drive each step. For engineering workflows, this means the agent can: search for datasheets, execute code, write and validate files, and deliver finished artifacts.

5.1 The Agentic Loop

The core pattern is a Reason → Act → Observe loop, also called ReAct (Reasoning + Acting). At each step the model produces either a tool call or a final response:

```
LOOP:
  1. THINK   — LLM reads system prompt + conversation history
               + tool results → produces reasoning (chain-of-thought)
  2. ACT     — LLM emits a tool_call with name + arguments
  3. OBSERVE — Tool executes; result appended to context
  4. REPEAT  — Until LLM emits a final text response (no tool call)

For our FIR agent the tools available are:
  web_search      — fetch Xilinx/Intel datasheets, IEEE papers
  code_runner     — execute C or Python in a sandbox
  file_generator  — produce .docx, .pdf, .verilog files
  memory_store    — retrieve/store user preferences, past designs
```

5.2 System Prompt Engineering — How We Tuned Our Agent

The quality of an AI agent's output is determined primarily by its system prompt. The system prompt injects domain context that the general model would not otherwise apply. For our Verilog & DSP agent the system prompt specifies:

```
SYSTEM PROMPT (key elements):

You are an expert DSP engineer, FPGA architect, and embedded C programmer.

Domain knowledge injected:
- FIR filters always have ALL poles at z=0 (trivial, N-1 poles)
- Symmetric coefficients → palindromic polynomial → zeros on unit circle
- ACC_WIDTH >= DATA_WIDTH + COEF_WIDTH + ceil(log2(N))
- DSP48E2 cascade: PCIN/PCOUT, USE_PATTERN_DETECT, PREG=1
- Q15 format: divide by 32768 to recover float value

Output requirements:
- Verilog must be synthesizable (no $display in RTL)
- C code must be C99, no external libraries unless asked
- Poles/zeros: state ALL N-1 poles at z=0 explicitly
- User is IEEE Senior Member — peer-level technical depth
```

▸ **Why domain context matters —** Without explicit domain instruction, a general-purpose LLM might omit the trivial poles at z=0, use non-synthesizable Verilog constructs, or generate C that depends on scipy. The system prompt makes these requirements non-negotiable by establishing them as baseline knowledge for the agent persona.

5.3 Conversation History — How the Agent Maintains State

Unlike a stateless API call, an agent maintains state by sending the full conversation history with every request. Each message pair (user prompt + assistant response) is appended to the messages array:

```
// JavaScript — how history is built in our agent
let history = [];

// Each send() call:
history.push({ role: 'user',      content: userMessage });
const response = await callClaudeAPI({ messages: history });
history.push({ role: 'assistant', content: response });

// The API receives the FULL history each time:
// [system prompt] + [user1, assistant1, user2, assistant2, ...]

// This means the agent 'remembers':
//   - Which coefficient set was established earlier
//   - Which architecture was chosen
//   - Which test results were already shown
//   - Your preferred code style and naming conventions
```

5.4 Tool Specialization — Focus Chips in Our Agent

Our DSP agent provides four focus chips (RTL/Verilog, DSP Math, C Code, FPGA/Synthesis) that append a context tag to each user message. This costs only a handful of tokens but significantly shifts how the model weights its response:

RTL/Verilog	Biases response toward synthesizable RTL patterns, named signal conventions, Vivado/Quartus attributes, and timing-closure advice.
DSP Math	Biases toward z-domain analysis, polynomial arithmetic, numerical precision, and algorithm derivation with full mathematical notation.
C Code	Biases toward C99 portable code, fixed-point patterns, no BLAS/scipy, explicit memory layout, and embedded-system idioms.
FPGA/Synthesis	Biases toward resource estimation, DSP slice mapping, timing constraints, and post-synthesis verification advice.

6. AI-Assisted Workflows — Side-by-Side Comparison

This section shows exactly how each task from Part I changes when an AI agent is in the loop. The comparison is deliberately honest: AI accelerates mechanical tasks dramatically but requires the engineer to specify correctly and verify critically.

6.1 Task: Computing Poles & Zeros

Without AI (Part I approach)	With AI Agent

Time: 2–4 hours **Steps:** Derive polynomial by hand, write Durand-Kerner from scratch, debug C, run, interpret **Risk:** Implementation bugs in complex arithmetic, slow convergence if initial guesses poor **Output:** Correct if implementation verified	**Time:** 30–60 seconds **Prompt:** 'Compute zeros of 8-tap FIR h=[1024,2048,4096,8192,...] Q15. Show z-polynomial, list (re,im) pairs, verify palindromic property.' **Risk:** Agent may omit poles or use wrong normalization — engineer must verify **Output:** Code + output + interpretation, instantly

The critical difference. The engineer who wrote the Durand-Kerner code in Section 2 understands exactly why the initial guess radius is set to |h[N]/h[0]|^(1/N) × 1.2, why the offset angle of 0.4 radians avoids root clustering, and why 2000 iterations with tolerance 1e-12 is appropriate. That understanding is what allows them to verify the AI output — or catch it when it silently converges to the wrong roots.

6.2 Task: Writing Synthesizable Verilog

```
── EXAMPLE AI AGENT PROMPT ──────────────────────────────────

User: Write a parameterizable Verilog FIR module in transpose form.
      8 taps, 16-bit signed input, Q15 coefficients.
      Exploit coefficient symmetry to halve the DSP count.
      Add a comment on every non-obvious line.
      Confirm it maps to Xilinx DSP48E2 PCIN/PCOUT cascade.

── WHAT THE AGENT DOES ──────────────────────────────────────
1. Recognises 'transpose form' → uses acc[k] = x*h[k] + acc[k+1] pattern
2. Recognises 'symmetry' → pre-adder stage before multiply
3. Adds synthesis attributes: (* use_dsp = 'yes' *)
4. Sizes ACC_WIDTH using the formula from its system prompt
5. Returns complete module + port table + resource estimate

── WHAT YOU STILL MUST DO ───────────────────────────────────
- Verify the module instantiates cleanly in your top-level
- Run Vivado synthesis and confirm DSP48E2 inference in utilization
- Check timing report: the cascade should close at your target Fmax
- Simulate with the testbench from Section 3.4 and compare outputs
```

6.3 Task: Coefficient Generation in C

One of the highest-value AI agent use cases is generating boilerplate C that a senior engineer would write correctly but slowly. The agent produces a working firwin_hamming() in under 10 seconds; the engineer's job is to verify the window formula and check the Q15 quantization against MATLAB:

```
── AI-GENERATED (verified correct) ──────────────────────────

/* firwin_hamming: generate N-tap low-pass FIR, Hamming window  */
/* fc: normalized cutoff frequency in (0, 0.5)                  */
/* out_q15: output array of Q15 signed 16-bit coefficients      */
void firwin_hamming(int N, float fc, int16_t *out_q15) {
    int   M    = N - 1;          /* filter order */
    float sum  = 0.0f;
    float h[N];
    for (int k = 0; k < N; k++) {
        /* Hamming window */
        float w = 0.54f - 0.46f * cosf(2.0f*M_PI*k / M);
        /* Ideal sinc low-pass */
        float n = k - M/2.0f;
        float s = (n == 0.0f)
```

```
                ? 2.0f * fc
                : sinf(2.0f*M_PI*fc*n) / (M_PI*n);
        h[k] = w * s;
        sum += h[k];
    }
    /* Normalize so DC gain = 1.0, then quantize to Q15 */
    for (int k = 0; k < N; k++) {
        float norm = h[k] / sum;
        out_q15[k] = (int16_t)(norm * 32767.0f + 0.5f);
    }
}

── VERIFICATION CHECKLIST (engineer's responsibility) ──────────
☐ sum(out_q15) ≈ 32768 (DC gain ≈ 1.0 in Q15)?
☐ Symmetric: out_q15[k] == out_q15[N-1-k]?
☐ No coefficient exceeds ±32767 (no overflow in int16_t)?
☐ Frequency response at cutoff matches -6 dB spec?
```

6.4 Workflow Efficiency Comparison

Task	Manual (hrs)	With AI Agent (mins)	Speed-up	Verification effort
Poles/zeros computation	2–4 h	< 1 min	~200×	Medium — check all root magnitudes
Write direct-form Verilog	1–2 h	2–3 min	~40×	High — simulate, check synthesis
Write transpose-form Verilog	2–3 h	3–5 min	~40×	High — verify DSP mapping in report
firwin coefficient generator (C)	30–60 min	< 1 min	~60×	Medium — compare to MATLAB/scipy
Fixed-point overflow analysis	30 min	2 min	~15×	Low — formula is deterministic
Testbench generation	1–2 h	2–3 min	~40×	Medium — check stimulus and assertions
Frequency response table (C)	45 min	1 min	~45×	Low — spot-check known frequencies
Full DOCX design document	4–8 h	15–20 min	~25×	Medium — review content accuracy

▸ **The verification principle —** Speed-up figures are only realised if the engineer verifies the output. An unverified AI-generated Verilog module that fails timing or produces wrong output at synthesis costs more time than writing it manually. The engineer's role shifts from author to technical reviewer — which requires deeper understanding, not less.

7. Building the Verilog & DSP Agent — Implementation

This section documents the exact architecture of the AI agent built alongside this document. It is a self-contained HTML/JavaScript application that calls the Anthropic Claude API directly from the browser — no backend server required. Understanding its implementation gives you a template for building domain-specialized agents for other engineering tasks.

7.1 API Call Structure

Every user message triggers a single POST to the Anthropic /v1/messages endpoint. The payload includes the full conversation history, the domain-specialized system prompt, and a token budget:

```
const response = await fetch('https://api.anthropic.com/v1/messages', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    model:      'claude-sonnet-4-6',
    max_tokens: 1500,
    system:     SYSTEM_PROMPT,   // domain specialization
    messages:   history          // full conversation array
  })
});

const data    = await response.json();
const reply   = data.content
                  .filter(b => b.type === 'text')
                  .map(b => b.text)
                  .join('');

// Append to history for next turn
history.push({ role: 'assistant', content: reply });
```

7.2 System Prompt — Complete Text

The full system prompt that specializes the agent for DSP/Verilog/C work:

```
You are an expert DSP engineer, FPGA architect, and embedded C programmer
specializing in:
- FIR/IIR filter theory (z-transform, poles, zeros, group delay)
- Verilog/SystemVerilog RTL design (direct-form, transpose-form FIR,
  DSP48E2 cascade mapping, synthesis attributes)
- C programming for embedded DSP (fixed-point arithmetic, Q-format,
  coefficient generation, filter implementations without BLAS/scipy)
- Fixed-point math (Q15, Q31, overflow detection, saturation, bit-growth)
- FPGA synthesis constraints (Xilinx, Intel, timing closure)

Guidelines:
- For poles/zeros: state that all N-1 poles are at z=0 explicitly.
  List zeros as complex (re, im) pairs with |z| and angle in degrees.
- For Verilog: produce synthesizable RTL only. Comment every
  non-obvious line. Name signals after their function.
- For C code: C99, no external libraries unless asked. Use fixed-point.
- The user is an IEEE Senior Member — peer-level technical depth.
```

7.3 Quick Prompt Library — Pre-Built Queries

The agent sidebar contains eight engineered prompts that cover the most common DSP engineering tasks. Each prompt is tuned to elicit a complete, verified response:

Poles/zeros — 8-tap	Compute zeros of 8-tap FIR h=[1024,2048,4096,8192,8192,4096,2048,1024] Q15. Show z-domain polynomial, list (re,im) pairs, verify palindromic property, state all poles explicitly.
C: freq response	Write C99 to compute frequency response magnitude at M points using direct DFT evaluation. No BLAS. Print table of normalized frequency vs magnitude vs dB.
Verilog: transpose form	Write parameterizable Verilog FIR, transpose form, pipeline registers on accumulator chain. Explain DSP48E2 PCIN/PCOUT cascade mapping. Synthesis-ready.
Fixed-point overflow	Derive ACC_WIDTH for 16-bit × 16-bit FIR, N taps. Write C function detecting saturation before and after accumulation. Show maximum value calculation.
C: firwin Hamming	Write firwin_hamming(N, fc) generating N low-pass FIR coefficients, Hamming window, returning Q15 int16_t array. Include normalization step.
Verilog testbench	Write Verilog testbench for 8-tap FIR. Impulse test, sine at passband and stopband frequencies. VCD dump. $monitor for every valid output.
Group delay & zeros	Explain group delay of linear-phase FIR, N=8. Derive constant group delay = (N-1)/2 samples. Show how it appears in pole-zero diagram.
C: .coe file runner	Write C99 that reads Xilinx .coe coefficient file, runs FIR sample-by-sample on float input array, writes output to CSV. Include main() with test data.

7.4 Extending the Agent for Other Engineering Tasks

The same architecture generalizes immediately to other domains by changing three things: the system prompt, the quick prompt library, and the focus chip labels. Below are extension patterns for adjacent engineering domains:

Domain	System prompt additions	Quick prompts to add	Focus chips
EMV / ISO 8583	DUKPT key derivation, PIN block formats, MAC calculation, TLV parsing rules	DUKPT KSN derivation in C, ISO 8583 field bitmap parser, EMV TLV decoder	Crypto, ISO 8583, EMV, PCI
PCI PIN compliance	PCI PIN Security v3 control objectives, HSM command sets, ZPK/ZMK formats	HSM command sequence for PIN translation, DUKPT→ZPK C code, PCI PIN audit checklist	HSM, DUKPT, ZPK, Audit
Android SoftPOS	Android Keystore API, Rubean SDK integration, NFC reader lifecycle, terminal session management	SoftPOS enrollment flow in Java, NFC reader state machine in Kotlin, SDK error code reference	NFC, Keystore, Rubean, AMS

Signal processing	FFT algorithm selection, windowing functions, spectral leakage, decimation	Radix-2 FFT in C, CIC decimator Verilog, polyphase filter bank	FFT, Window, Decimate, CIC

8. Best Practices — Working Effectively with a DSP Agent

An AI agent is a force multiplier, not an authority. The practices below reflect how an experienced DSP engineer integrates AI assistance into a rigorous engineering workflow without sacrificing correctness or accountability.

8.1 Prompt Engineering for DSP Tasks

- **Be specific about number formats.** Always state Q15, Q31, or float. An agent that guesses will often produce code that is numerically correct but in the wrong format for your pipeline.
- **State the toolchain.** 'Xilinx Vivado 2024.1, UltraScale+ xcku5p' changes synthesis attribute advice, resource naming, and timing constraint syntax.
- **Specify what NOT to use.** 'No scipy, no BLAS, no complex.h' prevents the agent from taking shortcuts that are unavailable in your embedded target.
- **Request verification alongside the code.** Append 'Include a verification step that checks DC gain equals sum(h)/32768' to get testable assertions built in.
- **Ask for the derivation, not just the result.** 'Show your working for the ACC_WIDTH formula' lets you catch wrong assumptions before they become synthesis errors.
- **Use conversation continuity.** After getting the Verilog module, say 'Now write the testbench for the module you just produced' — the agent has the interface in context and will generate matching port names automatically.

8.2 Verification Protocol — Never Skip These

1. **Poles/zeros output:** Verify all |z| values. For symmetric coefficients, all must be exactly 1.0 (within floating-point tolerance ~1e-10). Any deviation indicates a bug in the root-finder or coefficient normalisation.
2. **Verilog synthesis:** Run through Vivado or Quartus. Check the utilisation report for DSP slice count (should be N/2 with symmetry, N without). Check timing report — critical path should be one DSP slice, not an LUT adder chain.
3. **Fixed-point impulse response:** Feed a unit impulse (x=32768, then zeros) through the C filter. Output samples should equal h[0], h[1], ..., h[N-1] within Q15 rounding. Any mismatch indicates a shift-register direction error or index bug.
4. **Frequency response spot-check:** Evaluate |H(e^jw)| at omega=0 (should equal sum(h)/32768 ≈ 0.969) and at omega=pi (should be ~0.0 for this LP filter). These two checks catch the majority of implementation errors.

5. **Compare AI output to manual output:** Run the C code from Section 2 (Durand-Kerner) and the AI-generated zero finder on the same coefficient set. The zero locations should match to at least 6 significant figures.

8.3 What AI Agents Do Well vs. Poorly in DSP

Category	AI does well	AI does poorly / needs verification
Root finding	Implementing standard algorithms (Durand-Kerner, Jenkins-Traub)	Choosing appropriate convergence tolerance for your specific polynomial
Verilog RTL	Boilerplate, port declarations, synthesis attributes, testbenches	Timing closure decisions, custom FPGA floorplanning, clock domain crossing
C DSP code	Standard filter structures, window functions, DFT evaluation	Numerical stability edge cases, platform-specific alignment/endian issues
Documentation	Producing comprehensive well-formatted technical documents	Knowing which specifications changed since its training cutoff — always verify
Math derivation	Applying standard identities (palindromic polynomial, DFT symmetry)	Novel derivations — always cross-check against a textbook or MATLAB

9. Quick Reference — Formulae, Tables & Checklists

9.1 Essential DSP Formulae

```
FIR transfer function:
  H(z) = SUM h[k]·z^-k  =  P(z) / z^(N-1)
         k=0..N-1

Frequency response (unit circle):
  H(e^jw) = SUM h[k]·e^-jwk

Magnitude:   |H(e^jw)| = sqrt(Re² + Im²)
Phase:        phi(w) = atan2(Im, Re)
Group delay:  tau(w) = (N-1)/2  [samples, symmetric FIR]

Accumulator width:  ACC >= DATA + COEF + ceil(log2(N))

DC gain:    H(1)  = SUM h[k] / Q_SCALE
Nyquist:    H(-1) = SUM h[k]·(-1)^k / Q_SCALE

Palindromic property (symmetric FIR):
  P(z) = z^(N-1)·P(1/z)  →  all zeros |z|=1
```

9.2 Verilog RTL Checklist

- **ACC_WIDTH:** ≥ DATA_WIDTH + COEF_WIDTH + ceil($\log_2$(N)). For N=8,16-bit: ≥35 bits. Use 40.
- **Shift direction:** Write tail-first (x_reg[N-1] ← x_reg[N-2] ← ... ← x_in) to avoid read-before-write in a single always block.
- **Signed arithmetic:** Declare all data paths as signed. Unsigned × signed = wrong result silently in Verilog.
- **Synthesis check:** DSP slice count = N (direct) or N/2 (with symmetry pre-adder). LUT-based multipliers indicate missing synthesis directive.
- **Timing:** Direct form: critical path = 1 mult + $\log_2$(N) adders. Transpose: 1 mult + 1 adder. Use transpose for Fmax > 250 MHz.
- **Valid flag:** y_valid must be delayed by same number of pipeline stages as y_out. Mismatch corrupts downstream data.

9.3 Agent Prompt Templates

Zeros (generic)	Compute zeros of N-tap FIR with coefficients h=[...] in Q15. Show numerator polynomial, list (re,im) pairs with |z| and angle. State all poles at z=0.
Verilog module	Write synthesizable Verilog FIR: [direct/transpose] form, N=[N] taps, DATA_WIDTH=[W], Q15 coefficients, symmetry pre-adder=[yes/no]. Add synthesis attributes for DSP48E2.
C coefficient gen	Write C99 firwin_[window](int N, float fc, int16_t *out) generating N-tap LP FIR, [window] window, Q15 output. Include DC normalization. No external libraries.
Frequency response	Write C99 fir_magnitude(double omega, const int16_t *h, int N) returning |H(e^jw)|. Print table: 0 to pi in M steps, normalized freq, magnitude, dB.
Testbench	Write Verilog testbench for module [name] with ports [port list]. Tests: impulse, step, sine at [f1] and [f2]. VCD dump. $monitor every valid output sample.

9.4 References

Proakis & Manolakis	Digital Signal Processing, 4th ed. — Chapter 8: FIR filter design
Oppenheim & Schafer	Discrete-Time Signal Processing, 3rd ed. — z-transform and filter structures
Xilinx PG149	FIR Compiler v7.2 Product Guide — DSP48E2 cascade implementation
Xilinx UG579	UltraScale Architecture DSP Slice Reference — PCIN/PCOUT cascade ports
Intel AN 306	Implementing FIR Filters Using Intel FPGA DSP Blocks
IEEE Std 1800-2017	SystemVerilog Language Reference Manual
corebaseit.com	Vincent Bevia — FPGA, DSP & Payments Engineering