

Bridging the Verification Gap in DSP Designs: A Case Study on LMS Adaptive Filter Datapath Cycle-Accurate Verification Using Generative AI and MATLAB DPIGEN

Kuan-Ting Chen

kuan-ting.chen@silabs.com

Silicon Labs, 400 W Cesar Chavez St, Austin, TX 78701, USA

Abstract- Digital Signal Processing (DSP) blocks are the cornerstone of modern communication systems, performing critical arithmetic operations that require meticulous verification to ensure error-free data flow. Typically, DSP algorithms are developed in MATLAB by system teams. These algorithms are then implemented in Verilog by RTL designers. The primary responsibility for the verification engineers is to ensure that the RTL implementation accurately reflects the MATLAB algorithm models. However, a verification gap arises due to the disparate languages used by the system and design teams.

This paper explores the use of generative artificial intelligence (Microsoft Copilot) and the MATLAB `dpigen` function to bridge this gap by generating SystemVerilog assertions as well as DPI (Direct Programming Interface) components directly from MATLAB functions. This method facilitates efficient verification through direct comparison between the MATLAB model output and the RTL design output.

Through a detailed case study of the Least Mean Squares (LMS) adaptive filter, where both the MATLAB functions and RTL Verilog modules are generated by Microsoft Copilot, this paper outlines a step-by-step process for performing cycle-accurate verification using `dpigen`. It showcases how this approach can effectively identify and resolve discrepancies between the MATLAB model and the RTL implementation. Furthermore, this paper demonstrates how Microsoft Copilot can be used to create SystemVerilog assertions for intermediate datapath signal checking, addressing the challenge of limited visibility into internal signals during DPI-based verification.

Notably, three bugs in the Microsoft Copilot generated RTL design were identified and rectified through this process, underscoring the practical effectiveness of using generative AI and MATLAB `dpigen` in DSP verification.

I. INTRODUCTION

DSP logic plays an important role in IoT chips. The system team typically develops their MATLAB models to study and evaluate the DSP algorithms. Once the algorithm is mature, the RTL designers implement the model in Verilog.

One of the traditional DSP verification approaches involves applying test vectors provided by the system team to both the MATLAB model and the RTL. The verification engineer then compares the results and identifies any mismatches. However, this method is limited in scalability and struggles to cover extensive design state spaces, leaving some bugs uncovered in the corner cases.

Another common approach requires verification engineers to build their own predictors in SystemVerilog based on the specifications. This method can be problematic due to potential misunderstandings of the specifications, which necessitate iterative revisions of the predictors. Besides, any changes in the algorithm by the system team require the verification engineers to manually update the predictors, making this approach time-consuming, error-prone, and inefficient.

To address these challenges, a more efficient and effective solution is to directly reuse the MATLAB model as a golden reference in the testbench. With MathWorks HDL Verifier™ `dpigen` function [1], SystemVerilog DPI components can be automatically generated from MATLAB functions, as illustrated in Figure 1. These DPI components serve as predictors in the testbench, enabling automated, cycle-accurate verification of the design under test (DUT). This method not only supports constraint randomization for stimulus, potentially uncovering corner case bugs, but also enhances the efficiency and reliability of the verification process.

In practice, one limitation of DPI-based datapath verification is its difficulty in debugging. DPI components operate as black boxes during simulation, where only the input and output arguments are visible. Intermediate results or internal variables are not accessible from outside the DPI components. This makes it hard to pinpoint the cause of any mismatches in the output, which are often due to bugs within the datapath that propagate the errors to the final outputs. Adding SystemVerilog assertions to check the values of internal signals can make debugging

easier. Generative AI can assist in writing such assertions for datapath checking, provided both the MATLAB function and the RTL are available.

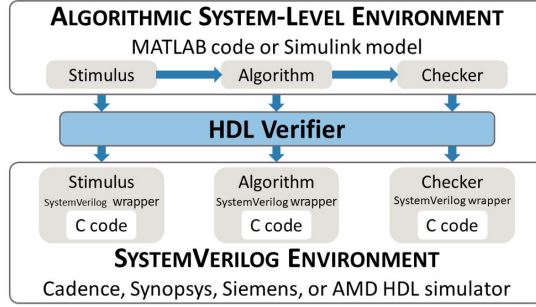


Figure 1. HDL Verifier™ dpigen function generates SystemVerilog DPI component from MATLAB function [2]

II. LMS ADAPTIVE FILTER DATAPATH VERIFICATION

This section detailed the step-by-step process for cycle-accurate datapath verification in DSP designs, using an LMS adaptive filter as an example. The LMS adaptive filter minimized the mean square error between a desired signal and the filtered signal by adjusting the filter coefficients iteratively.

Both the MATLAB function and RTL Verilog module were generated using Microsoft Copilot [3]. The MATLAB function served as the algorithm provided by the system team, which was used as a specification of the design as well as the golden reference model used in the verification. Additionally, SystemVerilog assertions were created to verify internal signals within the datapath. A testbench was used to demonstrate the LMS adaptive filter’s effectiveness in noise cancellation.

A. LMS Adaptive Filter MATLAB Model

The MATLAB function, generated by Copilot, acted as the design specification and golden reference model for verification. It is created based on the prompt: “Please give me one Least Mean Square adaptive filter MATLAB function named “lms_filter_one_cycle” which calculates only one clock cycle. The datapath signals should be quantized by M bits using built-in function floor as truncation. Inputs are x (input signal), d (desired signal), w (weight factor), mu (learning rate), M (number of bits for quantization), K (number of taps). Outputs are y (output signal), w_updated (updated weight factor), e (error)”. The resulting MATLAB function is shown in Figure 2.

Copilot

Certainly! Here's the MATLAB function `lms_filter_one_cycle` with line numbers included:

```

1. function [y, w_updated, e] = lms_filter_one_cycle(x, d, w, mu, H, K)
2. % Inputs:
3. % x - input signal (vector of length K)
4. % d - desired signal (scalar)
5. % w - weight factor (vector of length K)
6. % mu - learning rate (scalar)
7. % H - number of bits for quantization (scalar)
8. % K - number of taps (scalar)
9.
10. % Outputs:
11. % y - output signal (scalar)
12. % w_updated - updated weight factor (vector of length K)
13. % e - error (scalar)
14.
15. % Calculate the output signal
16. y = sum(w .* x);
17.
18. % Quantize the output signal
19. y = floor(y * 2^H) / 2^H;
20.
21. % Calculate the error
22. e = d - y;
23.
24. % Quantize the error
25. e = floor(e * 2^H) / 2^H;
26.
27. % Update the weight factor
28. w_updated = w + mu * e * x;
29.
30. % Quantize the updated weights
31. w_updated = floor(w_updated * 2^H) / 2^H;
32. end

```

Figure 2. lms_filter_one_cycle MATLAB function generated by Microsoft Copilot

To verify the accuracy of the algorithm, a testbench was generated using the prompt “Please give me a MATLAB testbench to demonstrate the generated function “lms_filter_one_cycle” in noise cancelling application. The desired signal is a sine wave with amplitude of 3 and input signal is the addition of the desired signal and a noise with amplitude of 1. Also plot the mean squared error over time.” The MATLAB testbench is shown in Figure 3.

```

1. % Parameters
2. fs = 1000; % Sampling frequency
3. t = 0:1/fs:1; % Time vector
4. A_desired = 3; % Amplitude of desired signal
5. A_noise = 1; % Amplitude of noise
6. f = 5; % Frequency of sine wave
7. mu = 0.01; % Learning rate
8. M = 8; % Number of bits for quantization
9. K = 10; % Number of taps (filter length)
10.
11. % Generate signals
12. d = A_desired * sin(2 * pi * f * t); % Desired signal (sine wave)
13. noise = A_noise * randn(size(t)); % Noise signal
14. x = d + noise; % Input signal (desired signal + noise)
15.
16. % Initialize weights
17. w = zeros(K, 1);
18.
19. % Initialize variables for storing results
20. y = zeros(size(t));
21. e = zeros(size(t));
22. mse = zeros(size(t));
23.
24. % Run the LMS filter for each sample
25. for n = 1:length(t)
26.     x_vec = x(n:-1:n-K+1); % Input vector for current sample
27.     [y(n), e(n)] = lms_filter_one_cycle(x_vec, d(n), w, mu, M, K);
28.     mse(n) = mean(e(n).^2); % Mean squared error
29. end
30.
31. % Plot results
32. figure;
33. subplot(3,1,1);
34. plot(t, d, 'b', t, x, 'r');
35. title('Desired Signal and Input Signal');
36. legend('Desired Signal', 'Input Signal');
37.
38. subplot(3,1,2);
39. plot(t, y, 'g');
40. title('Output Signal');
41. legend('Output Signal');
42.
43. subplot(3,1,3);
44. plot(t, mse, 'k');
45. title('Mean Squared Error');
46. legend('MSE');
47.
48. xlabel('Time (s)');
49. ylabel('Amplitude');

```

Figure 3. MATLAB testbench generated by Microsoft Copilot

The results, illustrated in Figure 4, highlighted the effectiveness of the generated MATLAB function in performing noise cancellation. The mean squared error (MSE), a metric used to measure the average squared difference between the output signal and the desired signal, showed a consistent reduction over time. This declining trend in MSE indicated that the adaptive filtering process was functioning as intended. As the filter adapted, the output signal converged closer to the clean sine wave, demonstrating that the generated MATLAB function successfully removed unwanted high-frequency noise components. Consequently, this function served as a reliable reference model for verifying DSP algorithms in hardware designs.

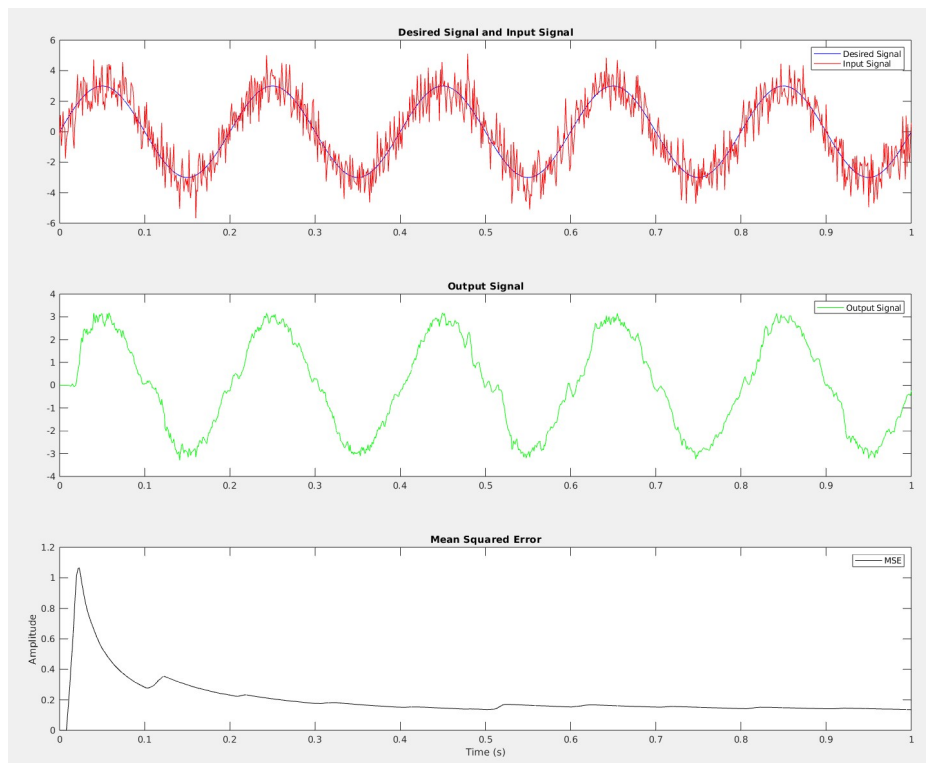
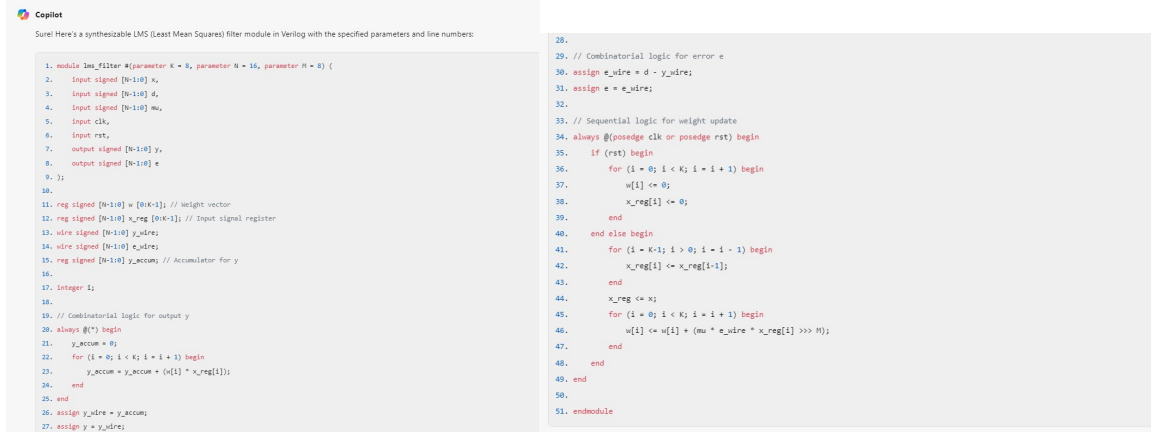


Figure 4. Effectiveness of the generated “lms_filter_one_cycle” function in noise cancellation

B. LMS Adaptive Filter RTL

The RTL Verilog module, also generated by Copilot, adheres to the prompt: “Please give me a synthesizable LMS Verilog module named “lms_filter” with K taps, where the data are N bits in width, and M out of N bits are for fractional bits. Inputs are x (input signal), d (desired signal), μ (learning rate), clk (clock), rst (reset) Outputs are y (output signal), e (error). Please pay attention to the bit width of the product in each multiplication. The weight vectors are sequential logics, and output signal y and error e are combinatorial logics.”

The generated Verilog code is shown in Figure 5, and it was designed to implement the LMS adaptive filtering algorithm in hardware.



```

1. module lms_filter #(parameter K = 8, parameter N = 16, parameter M = 8) (
2.     input signed [N-1:0] x,
3.     input signed [N-1:0] d,
4.     input signed [N-1:0] mu,
5.     input clk,
6.     input rst,
7.     output signed [N-1:0] y,
8.     output signed [N-1:0] e
9. );
10.
11. reg signed [N-1:0] w [0:K-1]; // weight vector
12. reg signed [N-1:0] x_reg [N-1:1]; // input signal register
13. wire signed [N-1:0] y_dire;
14. wire signed [N-1:0] e_dire;
15. reg signed [N-1:0] y_accum; // accumulator for y
16.
17. integer i;
18.
19. // Combinatorial logic for output y
20. always @(*) begin
21.     y_accum = 0;
22.     for (i = 0; i < K; i = i + 1) begin
23.         y_accum = y_accum + (w[i] * x_reg[i]);
24.     end
25. end
26. assign y_dire = y_accum;
27. assign y = y_dire;
28.
29. // Combinatorial logic for error e
30. assign e_dire = d - y_dire;
31. assign e = e_dire;
32.
33. // Sequential logic for weight update
34. always @(posedge clk or posedge rst) begin
35.     if (rst) begin
36.         for (i = 0; i < K; i = i + 1) begin
37.             w[i] <= 0;
38.         end
39.     end else begin
40.         for (i = K-1; i > 0; i = i - 1) begin
41.             w_reg[i] <= w_reg[i-1];
42.         end
43.         w_reg[0] <= w;
44.         for (i = 0; i < K; i = i + 1) begin
45.             w[i] <= w[i] + (mu * e_dire * x_reg[i] >> M);
46.         end
47.     end
48. end
49. endmodule

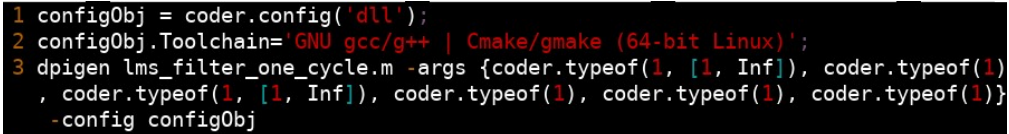
```

Figure 5. lms_filter Verilog module generated by Microsoft Copilot

C. Golden Reference Model as Predictor

The predictor was automatically created using the SystemVerilog DPI component of the MATLAB function “lms_filter_one_cycle”, generated via the dpigen command as shown in Figure 6. The command generated a shared library (.so file in a Linux environment), and a SystemVerilog package file containing the necessary function declarations.

The shared library allowed the MATLAB function to be called directly from the SystemVerilog environment, enabling seamless integration of the MATLAB algorithm into the hardware verification process. The SystemVerilog package file included the DPI function prototypes, making it straightforward to instantiate and use the MATLAB function within the SystemVerilog testbench.



```

1 configObj = coder.config('dll');
2 configObj.Toolchain='GNU gcc/g++ | Cmake/gmake (64-bit Linux)';
3 dpigen lms_filter_one_cycle.m -args {coder.typeof(1, [1, Inf]), coder.typeof(1),
  coder.typeof(1, [1, Inf]), coder.typeof(1), coder.typeof(1), coder.typeof(1)}
  -config configObj

```

Figure 6. dpigen command to generate DPI component from the MATLAB function

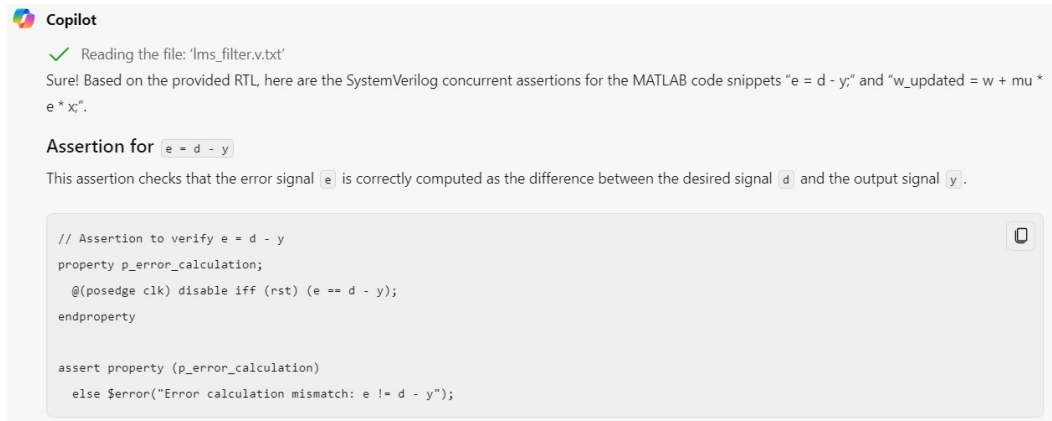
D. SystemVerilog Assertions

The assertions were helpful in checking intermediate signals in the datapath, which could unveil bugs earlier that would eventually propagate to the final output. By strategically placing assertions at key points in the RTL design, verification engineers could verify partial results during simulation, trace signal flow effectively, and isolate potential issues when mismatches occurred. This approach was particularly valuable in DPI-based verification, as it provided visibility into internal signal values, significantly enhancing the debugging process and improving overall verification efficiency.

Copilot was tasked with reading the MATLAB function and the RTL, and then generated two SystemVerilog concurrent assertions. The first assertion represented the combinatorial datapath for the MATLAB code “ $e = d - y$ ”, as shown in Figure 7. This assertion ensured that the error signal “ e ” was correctly computed as the difference between the desired signal “ d ” and the output signal “ y ”. The second assertion represented the sequential datapath for the MATLAB code “ $w_{\text{updated}} = w + \mu * e * x$ ”, as shown in Figure 8. This assertion verified that the weight

update logic was correctly implemented, ensuring that the updated weight “w_updated” was accurately calculated based on the previous weight “w”, the learning rate “mu”, the error signal “e”, and the input signal “x”.

The prompt given to Copilot was “*Can you write SystemVerilog concurrent assertions based on MATLAB code “e = d - y;” and “w_updated = w + mu * e * x;”? The assertions will be used to verify the RTL. Please identify the RTL signal names and use them in assertions.*”



Copilot

✓ Reading the file: 'lms_filter.v.txt'

Sure! Based on the provided RTL, here are the SystemVerilog concurrent assertions for the MATLAB code snippets “e = d - y;” and “w_updated = w + mu * e * x;”.

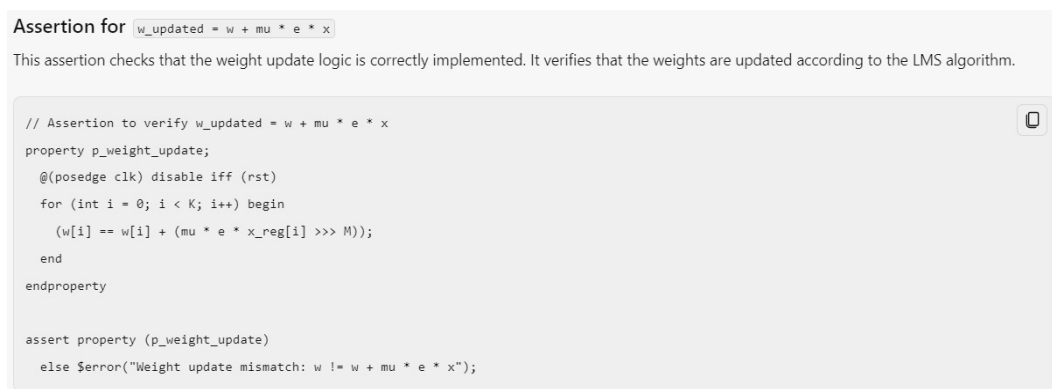
Assertion for `e = d - y`

This assertion checks that the error signal `e` is correctly computed as the difference between the desired signal `d` and the output signal `y`.

```
// Assertion to verify e = d - y
property p_error_calculation;
  @(posedge clk) disable iff (rst) (e == d - y);
endproperty

assert property (p_error_calculation)
  else $error("Error calculation mismatch: e != d - y");
```

Figure 7. SystemVerilog assertion for MATLAB code “e = d - y;” generated by Microsoft Copilot



Assertion for `w_updated = w + mu * e * x`

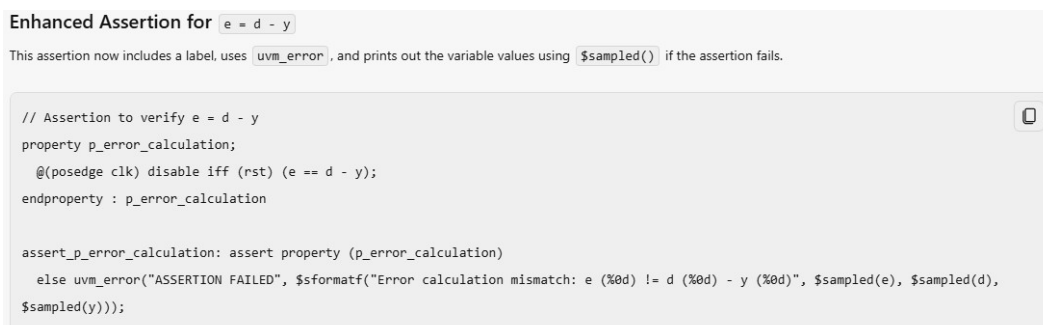
This assertion checks that the weight update logic is correctly implemented. It verifies that the weights are updated according to the LMS algorithm.

```
// Assertion to verify w_updated = w + mu * e * x
property p_weight_update;
  @(posedge clk) disable iff (rst)
  for (int i = 0; i < K; i++) begin
    (w[i] == w[i] + (mu * e * x_neg[i] >> M));
  end
endproperty

assert property (p_weight_update)
  else $error("Weight update mismatch: w != w + mu * e * x");
```

Figure 8. SystemVerilog assertion for MATLAB code “w_updated = w + mu * e * x;” generated by Microsoft Copilot

To improve the quality and debuggability of generated SystemVerilog assertions, the following prompt was provided to Copilot: “*Please do some enhancements including, 1. Use labels for assert statements 2. Use `uvm_error` instead of \$error 3. Use endproperty with end label 4. Print out variables value if assertion fail and use \$sampled() in value prints in action block*”. The enhanced assertions are shown in Figure 9 and Figure 10.



Enhanced Assertion for `e = d - y`

This assertion now includes a label, uses `uvm_error`, and prints out the variable values using `$sampled()` if the assertion fails.

```
// Assertion to verify e = d - y
property p_error_calculation;
  @(posedge clk) disable iff (rst) (e == d - y);
endproperty : p_error_calculation

assert_p_error_calculation: assert property (p_error_calculation)
  else uvm_error("ASSERTION FAILED", $formatf("Error calculation mismatch: e (%0d) != d (%0d) - y (%0d)", $sampled(e), $sampled(d), $sampled(y)));
```

Figure 9. Enhanced SystemVerilog assertion for MATLAB code “e = d - y;” generated by Microsoft Copilot

Enhanced Assertion for `w_updated = w + mu * e * x`

This assertion now includes a label, uses `uvm_error`, and prints out the variable values using `$sampled()` if the assertion fails.

```
// Assertion to verify w_updated = w + mu * e * x
property p_weight_update;
  @(posedge clk) disable iff (rst)
  for (int i = 0; i < K; i++) begin
    (w[i] == w[i] + (mu * e * x_reg[i] >>> M));
  end
endproperty : p_weight_update

assert_p_weight_update: assert property (p_weight_update)
  else uvm_error("ASSERTION FAILED", $sformatf("Weight update mismatch at index %0d: w[%0d] (%0d) != w[%0d] (%0d) + mu (%0d) * e (%0d) * x_reg[%0d] (%0d)", $sampled(i), $sampled(i), $sampled(w[i]), $sampled(i), $sampled(w[i]), $sampled(mu), $sampled(e), $sampled(i), $sampled(x_reg[i]))));
```

Figure 10. Enhanced SystemVerilog assertion for MATLAB code “`w_updated = w + mu * e * x;`” generated by Microsoft Copilot

The first assertion was straightforward and passed without issues, but the second, more complex assertion generated by Copilot had several issues. The for loop should not be inside the property, and the variables on the right-hand side should reflect the values from the previous clock cycle, requiring the use of the `$past` system function. Additionally, the simulator used context-sensitive expressions to determine the bit widths for arithmetic results, defaulting to the largest bit width of the variables involved. Therefore, the widths of variables in multiplication should be cast to $3*N$ bits to ensure correct arithmetic operation, and the product should be scaled by $2*M$ fractional bits.

After providing this feedback to Copilot, it generated an improved assertion, as shown in Figure 11. However, Copilot still did not correctly handle the bit width casting in multiplication. Therefore, human intervention was necessary to correct this. The revised assertions module is shown in Figure 12.

This experience highlighted the current limitations of generative AI in creating complex SystemVerilog assertions. While generative AI could assist in writing basic assertions, it often struggled with more intricate aspects, such as proper bit-width management, handling precision, and even correct syntax. As a result, verification engineers must still apply their expertise to refine and validate the generated assertions for correctness. However, they could leverage AI-generated assertions as a foundation to accelerate the development process. This approach enabled engineers to save valuable time, allowing them to focus on refining assertions rather than starting from scratch, ultimately enhancing the efficiency and effectiveness of the verification workflow.

Enhanced Assertion for `w_updated = w + mu * e * x`

This assertion now uses a generate block instead of a for loop, uses `$past` to represent variables in the previous clock cycle, scales the product by $2M$ bits, and includes bit width type casting to make `mu`, `e`, and `x_reg` $3N$ bits wide.

```
// Assertion to verify w_updated = w + mu * e * x
generate
  genvar i;
  for (i = 0; i < K; i++) begin : gen_weight_update
    property p_weight_update;
      @(posedge clk) disable iff (rst)
      (w[i] == $past(w[i]) + (($cast(3*N, mu) * $cast(3*N, e) * $cast(3*N, x_reg[i])) >>> (2*M)));
    endproperty : p_weight_update

    assert property (p_weight_update)
      else uvm_error("ASSERTION FAILED", $sformatf("Weight update mismatch at index %0d: w[%0d] (%0d) != w[%0d] (%0d) + mu (%0d) * e (%0d) * x_reg[%0d] (%0d)", $sampled(i), $sampled(i), $sampled(w[i]), $sampled(i), $sampled($past(w[i])), $sampled(mu), $sampled(e), $sampled(i), $sampled(x_reg[i]))));
    end
  end
endgenerate
```

Figure 11. Improved SystemVerilog assertion for MATLAB code “`w_updated = w + mu * e * x;`” generated by Microsoft Copilot with feedback

```

1 module lms_assertions #(parameter N = 16, parameter M = 8, parameter K = 4) (
2   input signed [N-1:0] d,
3   input signed [N-1:0] y,
4   input signed [N-1:0] e,
5   input signed [N-1:0] mu,
6   input signed [N-1:0] x_reg [0:K-1],
7   input signed [N-1:0] w [0:K-1],
8   input clk,
9   input rst
10 );
11
12 // Assertion to verify e = d - y
13 property p_error_calculation;
14   @(posedge clk) disable iff (rst) (e == d - y);
15 endproperty : p_error_calculation
16
17 assert_p_error_calculation: assert property (p_error_calculation)
18   else `uvm_error("ASSERTION FAILED", $sformatf("Error calculation mismatch: e (%0d) != d (%0d) - y (%0d)", $sampled(e), $sampled(d), $sampled(y)))
19 ;
20
21 // Assertion to verify w_updated = w + mu * e * x
22 generate
23   genvar i;
24   for (i = 0; i < K; i++) begin : gen_weight_update
25     property p_weight_update;
26       @(posedge clk) disable iff (rst)
27       (w[i] == $past(w[i]) + (((3*N)'($past(mu)) *
28         (3*N)'($past(e)) *
29         (3*N)'($past(x_reg[i]))) >>> (2 * M)));
30     endproperty : p_weight_update
31
32     assert property (p_weight_update)
33       else `uvm_error("ASSERTION FAILED", $sformatf("Weight update mismatch at index %0d: w[%0d] (%0d) != w[%0d] (%0d) + mu (%0d) * e (%0d) * x_reg
34 [%0d] (%0d)", $sampled(i), $sampled(i), $sampled(w[i]), $sampled(i), $sampled($past(w[i])), $sampled($past(mu)), $sampled($past(e)), $sampled(i), $sa
35 mpled($past(x_reg[i]))));
36   end
37 endgenerate
38 endmodule

```

Figure 12. Revised lms assertions module

E. Testbench

The testbench instantiated an eight-tap LMS adaptive filter with a learning rate of 0.125. The input and output data signals were 18 bits wide with 16 bits representing the fractional part. The desired signal was a 2 MHz sinusoidal wave. The input stimulus was a noisy sinusoidal wave composed of the desired signal and higher frequency sinusoidal waves of 20 MHz and 50 MHz. The testbench code is shown in Figure 13.


```

1 import uvm_pkg::*;
2 import lms_filter_one_cycle_dpi_pkg::*;
3
4 module tb_lms_filter;
5
6     // Parameters
7     parameter N = 18;
8     parameter M = 16;
9     parameter K = 8;
10    parameter integer CLK_PERIOD = 10;
11    parameter real PI = 3.141592653589793;
12
13    // Inputs
14    reg clk;
15    reg rst;
16    reg signed [N-1:0] x;
17    reg signed [N-1:0] d;
18    reg signed [N-1:0] mu;
19
20    // Outputs
21    wire signed [N-1:0] y;
22    wire signed [N-1:0] e;
23
24    // Instantiate the lms_filter module
25    lms_filter #(K, N, M) dut (
26        .clk(clk),
27        .rst(rst),
28        .x(x),
29        .d(d),
30        .mu(mu),
31        .y(y),
32        .e(e)
33    );
34
35    // Clock generation
36    initial begin
37        clk = 0;
38        forever #(CLK_PERIOD/2) clk = ~clk; // 10 ns period clock
39    end
40
41    // Stimulus process
42    real sine_wave, noise, noisy_sine, clean_sine;
43
44    initial begin
45        // Initialize inputs
46        rst = 1;
47        x = 0;
48        d = 0;
49        mu = 18'h2000; //0.125 in Q18.16 format
50
51        // Wait for the global reset
52        #(2 * CLK_PERIOD);
53        rst = 0;
54
55        // Apply test vectors
56        for (int i = 0; i < 1000; i = i + 1) begin
57            @(posedge clk);
58            // Generate clean sine wave and noisy sine wave
59            sine_wave = sin(2 * PI * i / 50);
60            noise = 0.2 * sin(2 * PI * i / 5) + 0.2 * sin(2 * PI * i / 2); // Higher frequency noise
61            noisy_sine = sine_wave + noise;
62            clean_sine = sine_wave;
63
64            // Convert to fixed-point representation
65            x = noisy_sine * (1 << M);
66            d = clean_sine * (1 << M);
67        end
68
69        // End simulation
70        #500;
71        $display("Finished Running Simulation");
72        $finish;
73    end
74
75    real x_dpi [K];
76    real d_dpi;
77    real w_prev_dpi [K];
78    real mu_dpi;
79    real y_dpi;
80    real w_updated_dpi [K];
81    real e_dpi;
82   chandle objhandle=null;
83    real y_temp;
84    real w_updated_temp [];
85    int w_updated_size_temp;
86    real e_temp;

```



```

87
88 // Initialize DPI function
89 initial begin
90     objhandle=DPI_lms_filter_one_cycle_initialize(objhandle);
91 end
92
93 // Terminate DPI function
94 final begin
95     DPI_lms_filter_one_cycle_terminate(objhandle);
96 end
97
98 // Drive DPI function inputs
99 always @(*) begin
100     mu_dpi = $itor(mu)/$itor(2**M);
101     d_dpi = $itor(d)/$itor(2**M);
102 end
103
104 genvar i;
105 generate
106     for (i=0; i<K; i++) begin
107         always @(posedge clk) begin
108             if(i==0) begin
109                 x_dpi [0] <= $itor(x)/$itor(2**M);
110             end
111             else begin
112                 x_dpi[i] <= x_dpi[i-1];
113             end
114         end
115     end
116     always @(posedge clk or posedge rst)begin
117         if(rst) begin
118             w_prev_dpi[i] <= 0;
119         end else begin
120             w_prev_dpi[i] <= w_updated_dpi[i];
121         end
122     end
123 end
124 endgenerate
125
126 // Call the MATLAB DPI at negedge of clock
127 always @(negedge clk) begin
128     DPI_lms_filter_one_cycle_output1(objhandle,x_dpi,d_dpi,w_prev_dpi,mu_dpi,M,K,y_temp,w_updated_size_temp,e_temp);
129     w_updated_temp=new(w_updated_size_temp);
130     DPI_lms_filter_one_cycle_output2(w_updated_temp);
131     y_dpi=y_temp;
132     w_updated_dpi=w_updated_temp;
133     e_dpi=e_temp;
134 end
135
136 // Compare the RTL output against MATLAB function output
137 always @(posedge clk) begin
138     check_signal(y, $rtoi(y_dpi * 2**M), "y");
139     check_signal(e, $rtoi(e_dpi * 2**M), "e");
140 end
141
142 function check_signal(int actual_signal, int expected_signal, string signal_name);
143 if(actual_signal != expected_signal)
144     $uvm_error("ERROR", $sformatf("%s mismatch: Actual %d, Expected %0d", signal_name, actual_signal, expected_signal));
145 endfunction : check_signal
146
147 // Bind the assertions
148 bind_lms_filter lms_assertions #(.N(N), .M(M), .K(K)) lms_assertions_inst (
149     d(d),
150     y(y),
151     e(e),
152     mu(mu),
153     x_reg(lms_filter.x_reg),
154     w(lms_filter.w),
155     clk(clk),
156     rst(rst)
157 );
158
159 endmodule

```

Figure 13. Testbench verifying lms filter RTL against DPI component

E. Debug

Three bugs were identified in the generated RTL. The first bug involved the incorrect assignment of a packed array “x” to an unpacked array “x_reg”. The second bug was related to incorrect bit widths of variables in multiplication operations. The bit width of the product should be the sum of the bit widths of the multiplier and multiplicand to prevent data overflow and maintain precision. In the RTL, the variable “y_accum” was only N bits wide, which was insufficient for accommodating the full precision of the multiplication results. It should have been 2*N bits wide to ensure that the result of multiplying two N-bit values was fully captured without truncation or overflow. Similarly, the product of “mu”, “e_wire”, and “x_reg” should be 3*N bits wide.

The third bug pertained to the weight update logic, specifically in the handling of the product of “mu”, “e_wire”, and “x_reg” signals, which represented the intermediate result of multiplying the learning rate, error signal, and input signal. Initially, this signal was truncated by only M bits, which was insufficient given that it has a precision of

2*M bits representing the fractional number. Proper truncation by 2*M bits was essential to accurately maintain the fractional bit calculations throughout the filtering process, ensuring the correct convergence of the adaptive filter.

Copilot was requested to fix the bugs using the prompt “*x_reg <= x; cannot be compiled. The error message is "Error. Cannot assign a packed type 'wire signed[N-1:0]' to an unpacked type 'reg signed[N-1:0]' [0:K-1]" y_accum is only N bit wide but (w[i] * x_reg[i]) is a result of 2*N bit wide The bit width of the product of mu * e_wire * x_reg[i] should be 3*N bit wide, but in your code it is only N bits wide, and should be scaled by 2*M fractional bits*”.

Upon receiving this feedback, Copilot was able to comprehend the issues which are illustrated in Figure 14, and promptly made the necessary corrections. The revised Verilog code reflecting these modifications is presented in Figure 15. Although mixing blocking and non-blocking assignments within the same always block was generally not considered good practice, the code compiled successfully and functioned correctly. This demonstrated that generative AI, like Copilot, could debug Verilog code based on human input and error messages from the simulators, effectively addressing specific issues. However, human oversight remained essential to ensure the overall integrity and functionality of the code.

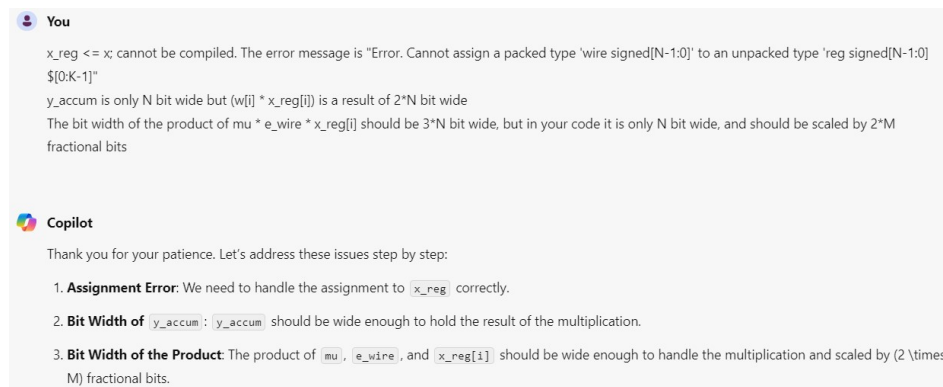


Figure 14. Copilot's response to feedback of identified bugs

```

1. module lms_filter #(parameter K = 8, parameter N = 16, parameter M = 8) (
2.     input signed [N-1:0] x,
3.     input signed [N-1:0] d,
4.     input signed [N-1:0] mu,
5.     input clk,
6.     input rst,
7.     output signed [N-1:0] y,
8.     output signed [N-1:0] e
9. );
10.
11. reg signed [N-1:0] w [0:K-1]; // Weight vector
12. reg signed [N-1:0] x_reg [0:K-1]; // Input signal register
13. wire signed [N-1:0] y_wire;
14. wire signed [N-1:0] e_wire;
15. reg signed [2*N-1:0] y_accum; // Accumulator for y
16. reg signed [3*N-1:0] product; // Product of mu, e_wire, and x_reg[i]
17.
18. integer i;
19.
20. // Combinatorial logic for output y
21. always @(*) begin
22.     y_accum = 0;
23.     for (i = 0; i < K; i = i + 1) begin
24.         y_accum = y_accum + (w[i] * x_reg[i]);
25.     end
26. end
27. assign y_wire = y_accum[2*N-1:M]; // Truncate to N bits
28. assign y = y_wire;
29.
30. // Combinatorial logic for error e
31. assign e_wire = d - y_wire;
32. assign e = e_wire;
33.
34. // Sequential logic for weight update
35. always @(posedge clk or posedge rst) begin
36.     if (rst) begin
37.         for (i = 0; i < K; i = i + 1) begin
38.             w[i] <= 0;
39.             x_reg[i] <= 0;
40.         end
41.     end else begin
42.         for (i = K-1; i > 0; i = i - 1) begin
43.             x_reg[i] <= x_reg[i-1];
44.         end
45.         x_reg[0] <= x;
46.         for (i = 0; i < K; i = i + 1) begin
47.             product = mu * e_wire * x_reg[i];
48.             w[i] <= w[i] + (product >>> (2*M));
49.         end
50.     end
51. end
52.
53. endmodule

```

Figure 15. Revised lms filter RTL, generated by Microsoft Copilot

With the updated RTL, the output variable “y” matched the DPI predictor output variable “y_dpi” on a cycle-by-cycle basis, as shown in Figure 16 and Figure 17. The revised RTL was rigorously verified against the MATLAB reference model, validating its functional correctness and demonstrating the effectiveness in noise cancellation.

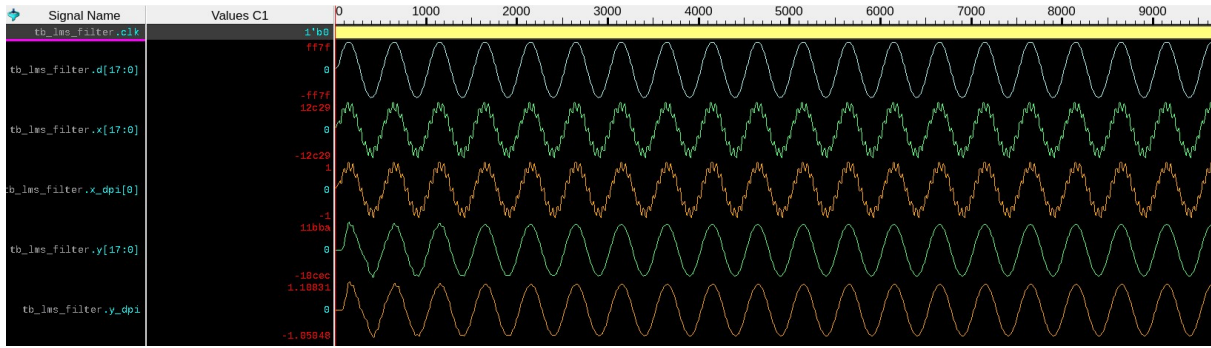


Figure 16. Analog waveform showing matching outputs of RTL and MATLAB function in every clock cycle

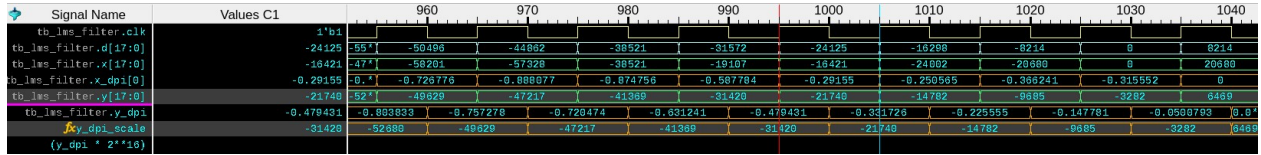


Figure 17. Digital waveform showing matching outputs of RTL and MATLAB function in every clock cycle

III. CONCLUSION

In DSP design workflows, system teams typically develop algorithms in MATLAB, which are then implemented in RTL based on these specifications. Reusing the MATLAB model as the predictor verification component is highly advantageous for verification engineers, as it significantly reduces both the time and effort required for predictor development. It also ensures that the RTL implementation adheres to the specifications. This paper demonstrates the application of the MATLAB dpigen function to automate the generation of SystemVerilog DPI components from MATLAB functions, enabling cycle-accurate verification of the design under test (DUT).

Furthermore, this paper demonstrates that SystemVerilog assertions can be created for intermediate signal checking in the datapath using generative AI, facilitating the early detection and debugging of any MATLAB-RTL mismatches. This approach is especially valuable in overcoming the limitations of the DPI, which often functions as a black box in verification due to the lack of visibility into internal signals. By incorporating assertions to check critical intermediate signals, this method offers greater transparency into datapath operations, ensuring that mismatches are identified and addressed earlier in the verification cycle.

Although the example used in this paper is a small DSP block, this approach is also applicable to complex datapath verification. Generative AI accelerates assertion development by analyzing designs and generating targeted assertions, minimizing the need for manual assertion development. By strategically placing assertions at critical points along the complex datapath, the root causes of MATLAB-RTL mismatches can be quickly narrowed down and isolated, leading to more efficient and effective verification workflows.

Generative AI also offers valuable assistance in DSP block design and verification. The experiment shows that generative AI can produce high-quality MATLAB code, potentially reducing the time system teams need for algorithm development. Moreover, generative AI can generate basic SystemVerilog assertions based on the MATLAB function. However, for more complex assertions, detailed and well-crafted prompts are necessary to achieve optimal results. While the quality of the generated code may not be mature enough for direct use, it provides a solid framework that can serve as a starting point for assertions development. This framework, however, still requires careful review and modifications by experienced verification engineers.

REFERENCES

- [1] dpigen Reference. <https://www.mathworks.com/help/hdlverifier/dpi-c-generation-for-matlab-code.html>
- [2] MathWorks, "SystemVerilog for Verification," MathWorks, [Online]. Available: <https://www.mathworks.com/discovery/systemverilog.html>. [Accessed: 27-Aug-2024].
- [3] Microsoft Copilot. Accessed 2024-10-21 [Online]. Available: <https://copilot.microsoft.com>