반도체디스플레이기술학회지 제23권 제1호(2024년 3월)
Journal of the Semiconductor & Display Technology, Vol. 23, No. 1. March 2024.

통신시스템용 등화기 모듈을 위한 UVM 기반 검증

문대원*·홍대기*†
*†상명대학교 전자정보시스템공학과

UVM-based Verification of Equalizer Module for Telecommunication System

Dae-Won Moon* and Dae-Ki Hong*†
*†Department of Electronic and Information and System Eng., SangMyung University

ABSTRACT

In the present modern day, as the complexity and size of SoC(System on Chip) increase, the importance of design verification are increasing, Therefore it takes a lot of time to verify the design. There is an emerging need to manage the verification environment faster and more efficiently by reusing the existing verification environment. UVM-based verification is a standardized and highly reliable verification method widely adopted and used in the semiconductor industry. This paper presents a UVM-based verification for the 4 tap equalizer module with a systolic array structure. Through the constraints randomization, it was confirmed that various test scenarios stimulus were generated. In addition, by verifying a simulation comparing the actual DUT outputs with the MATLAB reference outputs, the reuse and efficiency of the UVM test bench could be confirmed.

Key Words : UVM, VIP, Equalizer, Verification, Extended QRD-RLS

1. 서 론

유·무선통신시스템에서의 등화기의 성능은 전체 시스템의 성능에 중요한 영향을 미친다. 통신시스템 성능 향상을 위해 고속데이터 전송과 신호간섭을 주지 않는Binary-CDMA(Code Division Multiple Access) 모뎀 구현과 수백 Mbps 이상의 데이터 전송률을 실현하는 UWB (Ultra-Wideband) 무선 전송방식 중의 하나인 MB-OFDM(Multi-Band Orthogonal Frequency Division Multiplexin) 시스템의 SoC (System on Chip) 구현의 연구도 소개되었다[1-2].

특히 시변채널 환경에서는 짧은 시간에 효과적인 신호보상이 관건이 된다. 등화기는 어떤 시점에 대해서 참조신호와 실제 출력신호와의 오차신호의 제곱합인 비용함수를 최소화 해야만 한다. 이를 위해서 비용함수를 최소로 하게 하는 최적 weight vector를 결정하기 위해 다양한 알고리즘을 활용하게 된다. 최급 강하법에 근거를 둔 LMS(Least Mean Square) 알고리즘과 재귀적 최소 제곱법인 RLS(Recursive Least Square) 알고리즘이 대표적이다. LMS 알고리즘은 처리속도가 느린 반면, 계산부담이 현저히 작고 간단한 연산만 실현 가능하여 소규모 하드웨어에 효율적이다. 이에 반해 RLS알고리즘은 복소행렬을 처리하는데 있어서 계산부담이 크다는 단점이 있지만 처리속도가 빠르고 연산 정확도가 우수하여 MPU(Micro Processing Unit), DSP(Digital Signal Processing), FPGA(Field Programmable Gate Array) 등의 대규모 하드웨어에 실장되고 있다.

RLS알고리즘의 복잡한 행렬 연산과 곱셈기의 증가에 따른 구현 복잡도 문제를 줄이기 위하여 QRD(Quadratic Residue Decomposition)을 사용한 행렬의 삼각화(Triangularization)방식이 소개되었다[3-4].

또한, RLS 알고리즘의 계산부담을 고려한 실시간 처리와

†E-mail: hongdk@smu.ac.kr

병렬처리를 위하여 Givens 회전을 사용한 시스톨릭 어레이 구조가 제안되었다[5-6].

시스톨릭 어레이는 단순계산을 처리하는 cell을 일정한 규칙에 의해 배열하고, 계산에 요구되는 데이터를 파이프라인 형에 입력시켜 병렬계산을 하는 구조이고, 파이프라인 형은 RISC (Reduced Instruction Set Computer) 기반의 프로세서 설계 및 구현방식에도 사용된다[7].

입력 신호 행렬의 특정 엘리먼트를 없애기 위한 벡터 회전을 수행한다. 회전 각도의 계산은 삼각함수 연산을 필요로 하는데 이를 위하여 CORDIC(COordinate Rotation DIgital Computer) 알고리즘이 사용될 수 있다[8-9].

CORDIC 알고리즘을 사용하는 시스톨릭 어레이 구조 연산은 shift와 adder/subtracter를 사용하는 반도체로 효율적으로 구현될 수 있다.

이와 같은 CORDIC 시스톨릭 어레이 구조에서는 입력행렬의 QR 분해를 수행하는 RLS 알고리즘의 동작을 Vector mode CORDIC block과 Rotation mode CORDIC block을 사용한다[10].

본 논문에서는 등화기에서 사용되는 QRD-RLS알고리즘과 확장 QRD-RLS 알고리즘에 대하여 소개하고, 시스톨릭 어레이 구조의 PE(Processing Elements)인 Vector mode CORDIC block과 Rotation mode CORDIC block이 반대 방향으로 회전하는 것에 착안하여 위의 2개의 CORDIC block이 동시에 동작하는 구조를 설계하고, 이를 최신 반도체 설계 모듈 검증에 적용하고 있는 UVM(Universal Verification Methodology)을 이용하여 재사용성, 유연성, 확장성을 보장하기 위한 검증 방법을 다루고자 한다.

2. 등화기용 QRD-RLS 알고리즘

2.1 QRD-RLS 알고리즘

QRD-RLS(QR Decomposition based Recursive Least Mean Squares) 알고리즘이 고속 수신기에 많이 사용되기 위해서는 계산의 복잡도가 감소되어야 한다. 이 알고리즘에서 사용되는 표기는 다음과 같다.

$d(i)$: 송신되는 파일롯 신호
$w(n)$: 구하려는 필터 계수 벡터
$u(i)$: 수신된 파일롯 신호 벡터
λ^{n-i} : 과거 파일롯 신호의 영향이 감소되는 것을 나타내는 forgetting factor

여기에서 목표는 필터계수 벡터 $w(n)$ 을 구하는 것이며 이 벡터는 다음의 비용함수를 최소화하도록 결정되어야 한다.

$$\varepsilon = \| Q(n)\Lambda(n)d(n) - Q(n)\Lambda(n)A(n)w(n) \|^2 \quad (1)$$

이 식에서 $Q(n)$ 은 unitary 행렬이며, Λ 는 망각계수 λ 로 이루어진 행렬이다. 또한 $d(n)$ 은 송신된 파일롯 벡터이며 $A(n)$ 은 수신된 파일롯 신호로 만들어진 Toeplitz 행렬이다. 위의 비용함수가 최소가 되기 위해서는 다음의 식을 만족하여야 한다.

$$R(n)w(n) = P(n) \quad (2)$$

이 식에서 $R(n)$ 은 상삼각행렬이고 벡터 $P(n)$ 은 $Q(n)\Lambda(n)d(n)$ 의 윗부분이다. 상삼각행렬 $R(n)$ 을 구하기 위해 다음의 Givens 회전을 사용하여 구할 수 있다.

$$J_M(n)...J_2(n)J_1(n)Q'(N-1)\Lambda(n)A(N) = \begin{bmatrix} R(n) \\ 0 \\ O^T \end{bmatrix} \quad (3)$$

벡터 $P(n)$ 도 역시 Givens 회전을 사용하여 다음과 같이 구할 수 있다.

$$\begin{bmatrix} P(n) \\ V(n) \end{bmatrix} = J_M(n)...J_2(n)J_1(n) \begin{bmatrix} \lambda P(n-1) \\ \lambda V(n-1) \\ d^*(n) \end{bmatrix} \quad (4)$$

이와 같이 $R(n)$ $P(n)$ 을 구하였으므로 목표가 되는 벡터는 다음과 같이 구할 수 있다.

$$w(n) = R^{-1}(n)P(n) \quad (5)$$

이 식에서 상삼각행렬 $R(n)$ 의 역행렬을 구하는 계산의 복잡도를 줄이기 위해 다음 절에서 다루는 확장 QRD-RLS 알고리즘이 제안되었다.

2.2 확장 QRD-RLS 알고리즘

QRD-RLS 알고리즘에서 구한 $R(n)$ 과 $P(n)$ 을 update하는 식을 합하여 표현하면 다음과 같다.

$$\begin{bmatrix} R(n) & P(n) \\ 0 & \alpha(n) \end{bmatrix} = Q(n) \begin{bmatrix} \lambda R(n-1) & \lambda P(n-1) \\ u^T(n) & d(n) \end{bmatrix} \quad (6)$$

망각계수를 곱하여 다음의 상삼각행렬을 새로정의한다.

$$\tilde{R}(n) = \begin{bmatrix} \lambda R(n) & \lambda P(n) \\ 0 & \alpha(n) \end{bmatrix} \quad (7)$$

위의 행렬의 역 행렬은 다음과 같이 표현될 수 있다.

$$\tilde{R}^{-1}(n)=\begin{bmatrix}\lambda^{-1}R^{-1}(n) & -w(n)/\alpha(n)\\ 0 & 1/\alpha(n)\end{bmatrix} \quad (8)$$

직교변환 행렬인 $Q(n)$이 상삼각행렬 $R(n)$을 update하는 것과 같은 방법으로 하삼각행렬도 다음과 같이 update한다.

$$\begin{bmatrix}\tilde{R}^{-T}(n)\\ v(n)\end{bmatrix}=Q(n)\begin{bmatrix}\tilde{R}^{-T}(n-1)\\ 0\end{bmatrix} \quad (9)$$

확장 QRD-RLS 알고리즘은 상삼각행렬과 하삼각행렬의 두 개의 행렬이 다음과 같이 update된다.

$$\begin{bmatrix}\tilde{R}(n) & \tilde{R}^{-T}(n)\\ 0 & v(n)\end{bmatrix}=Q(n)\begin{bmatrix}\tilde{R}(n-1) & \tilde{R}^{-T}(n-1)\\ \tilde{u}^{T}(n) & 0\end{bmatrix} \quad (10)$$

위의 update 식에서 목표 벡터인 $w(n)$ 은 상삼각행렬 $\tilde{R}(n)$ 의 우측 아래 코너에 놓여있는 $\alpha(n)$ 과 하삼각행렬 $\tilde{R}^{-1}(n)$ 의 맨 아랫줄을 곱하여 얻어진다.

3. 확장 QRD-RLS 등화기용 시스톨릭 어레이 구조와 설계

3.1 구조

지난 절에서 기술된 확장 QRD-RLS 등화기는4 tap 경우의 update식은 Fig. 1과 같은 시스톨릭 어레이 구조로 구현될 수 있다.

Fig. 1의 시스톨릭 어레이 구조의 각각의 PE는 CORDIC 알고리즘을 사용하여 쉽게 구현될 수 다[10]. Fig. 1에서 원모양은 Vector mode CORDIC으로서 벡터의 직각 좌표가 입력되면 그 벡터의 각도와 크기가 계산된다. 네모 모양은 Rotation mode CORDIC으로서 벡터와 각도가 입력되면 각도만큼 회전된 후의 벡터의 좌표를 출력한다.

Fig. 1에서 R_{11} 의 Vector mode CORDIC PE가 벡터의 각도를 계산하여 첫 번째 행의 나머지 5개의 Rotation mode CORDIC PE들에 그 각도를 제공한다. Rotation mode CORDIC PE들은 Vector mode CORDIC PE로부터 제공받은 각도와 함께 수신 신호 입력을 가지고 내부 회전을 통해 결과 값을 출력하고, 아래의 두 번째 행의 CORDIC PE들의 입력으로 전달한다. 세 번째 행은 두 번째 행의 CORDIC PE들로부터 계산된 좌표 값을 입력으로 받고, 네 번째 행은 세 번째 CORDIC PE들로부터 계산된 좌표 값을 입력으로 받는다.

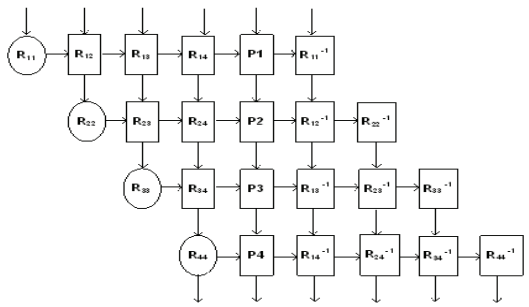


Fig. 1. Block diagram of Extended Systolic Array Architecture for 4 tap Equalizer.

Vector mode와 Rotation mode의 CORDIC PE는 모두 16 step의 회전을 통해 결과 값을 출력한다고 가정하면, Vector mode R_{11} 과 Rotation mode R_{12} 의 16 step 회전방향은 정확히 반대가 된다. 이와 같은 아이디어는 주파수 옵셋 동기화기에서도 사용되고 있다[11].

따라서, R_{11} Vector mode CORDIC PE는 각도를 계산하여 첫 번째 행의 5개 Rotation mode CORDIC PE들에 제공할 필요가 없으며 5개 Rotation mode CORDIC PE들은 R_{11} 의 16 step의 회전방향을 모니터링 하면서 반대 방향으로 회전하면서 벡터 값을 얻을 수 있게 된다.

Fig. 2는 16 step(C0 ~ C15)의 회전을 가지는 Vector mode CORIC R_{11} PE와 Rotation mode CORDIC R_{12} PE의 block이다.

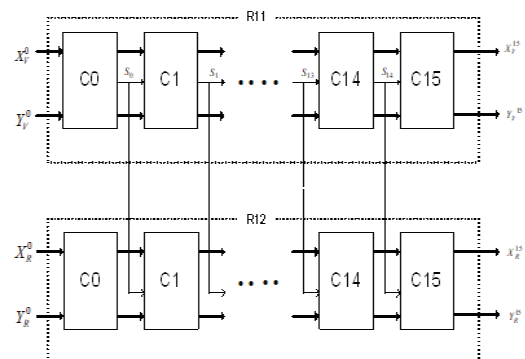
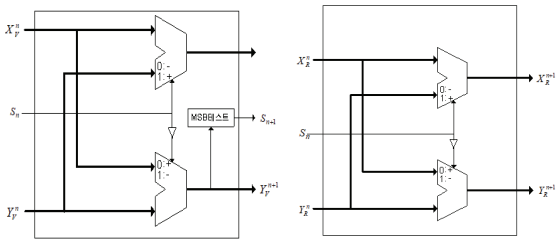


Fig. 2. Block diagram of R_{11} (16 step vector mode CORDIC PE) and R_{12} (16 step rotation mode CORDIC PE) Structure.

Fig. 3의 (a)는 Vector mode CORIC R_{11} PE의 내부 구조로서, 각도 update블록이 없는 벡터 update 블록과 회전방향 결정 블록으로만 구성된 회로를 보여 주고 있고, Fig. 3의 (b)는 Rotation mode CORIC R_{12} PE의 내부 구조로서, 각도 update블록과 회전방향결정 블록이 없는 벡터 update 블록으로만 구성된 회로를 보여 주고 있다



(a) Cn of R_{11} (b) Cn of R_{12}

Fig. 3. Structure of R_{11} and R_{12}.

3.2 설계

확장 QRD-RLS 등화기용 4-by-4시스톨릭 어레이 구조에 4개의 송신 신호를 입력하는 경우 총 소요 clock을 계산해 보기로 한다.

시스톨릭 어레이 구조의 같은 행의 1개의 Vector mode CORDIC PE와 나머지5개의 Rotation mode CORDIC PE가 동시에 회전하므로 같은 clock에 계산과 동작이 진행되고, 다음 clock에 아래 행들의 CORDIC PE로 벡터 좌표 값을 전달한다. 따라서 동작에 필요한 전체 clock 수는 Table 1과 같다.

Table 1에서 보듯이 총 7 clock에 전체 계산이 완료 됨을 볼 수 있다.

Table 1. Number of clock for systolic array in CORDIC Structure (4 tap equalizer, 4 input length)

4 3 2 1	4 3 2 1	4 3 2 1	4 3 2 1	4 3 2 1	4 3 2 1		
	5 4 3 2	5 4 3 2	5 4 3 2	5 4 3 2	5 4 3 2	5 4 3 2	
		6 5 4 3	6 5 4 3	6 5 4 3	6 5 4 3	6 5 4 3	6 5 4 3
			7 6 5 4	7 6 5 4	7 6 5 4	7 6 5 4	7 6 5 4

위의 Table 1에서 보듯이 7 clock이 지나면 식 (5)의 계산에 필요한 P(n) 과 $R^{-1}(n)$ 을 얻을 수 있다.

Fig. 1을 가지고 설명하면, P(n) 은 P1, P2, P3, P4로 구성된 열 벡터 이고 $R^{-1}(n)$ 은 P(n) 의 우측 4-by-4 하삼각행렬을 transpose하여 도출된다. 그리하여 P(n) 과 $R^{-1}(n)$ 을 도출하였으므로 식 (5)를 이용하여 등화기의 필터계수 w(n) 을 구할 수 있도록 설계한다.

확장 QRD-RLS 등화기용 시스톨릭 어레이 구조와 설계를 통해 등화기의 필터 계수를 찾는 방식을 알아 보았다. 다음 절에서는 3절에서 설계한 DUT를 최신 반도체 검증 방법인 UVM을 이용하여 재사용 가능한 테스트벤치를 구축하고, MATLAB을 이용한 Reference data와 실제 DUT 의Actual data를 비교하여 UVM 기반의 테스트벤치의 확장성과 유연성을 알아보고자 한다.

4. UVM 검증환경 테스트벤치 구현

4.1 확장 QRD-RLS 등화기용 UVM VIP ARCHITECTURE

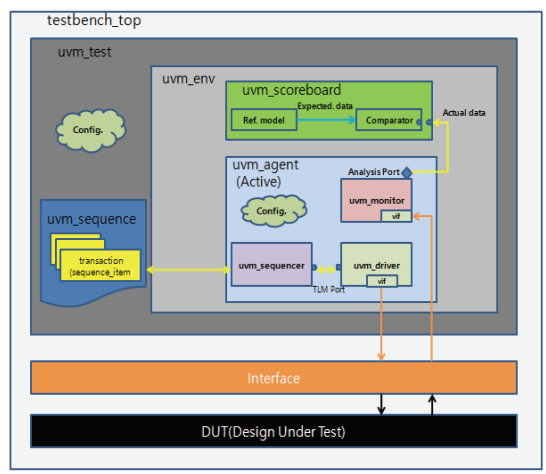


Fig. 4. Extended QRD-RLS Equalizer UVM VIP Block diagram.

Fig. 4는 확장 QRD-RLS Equalizer를 검증하기 위해 구축한 UVM 기반의 테스트벤치 아키텍쳐를 보여주고 있다. 테스트벤치 top은 크게 3가지 블록으로 구성되어 있다.

첫 번째로 uvm_* 으로 시작하는 UVM class를 기반으로 재사용 가능한 components가 있다. 두 번째로 UVM components와 DUT 간의 인터페이스 역할을 담당하는 interface가 있다. 마지막으로 검증하고자 하는 설계 DUT가 있다. 노랑색 화살표는 UVM components간의 통신에 사용되는 data인 transaction을 가리킨다. 주황색 화살표는 interface signal이며, 검정색 화살표는 DUT signal이다. 또한, 구름모양은 환경구성에 필요

한 정보를 해당되는 위치에서 configuration database에서 설정하는 표시이다. uvm_test에 있는 config.는 테스트벤치에서 사용되는 virtual interface를 설정하는 부분이고, uvm_agent에서의 config.는 agent를 구성하는 sequencer/driver/monitor를 갖는 구성을 위해 설정하는 부분이다.

위의 구축한UVM 테스트벤치를 구성하는 각 블록에 대해서 다음 절에서 다루고자 한다.

4.2 SEQUENCE ITEM

4 tap 등화기에 사용되는 4-by-4시스톨릭 어레이 구조를 검증하기 위해서는 DUT를 drive하는 stimulus가 필요하며, uvm sequence item class에서는 stimulus 생성을 위해 필요한 transaction을 정의한다.

transactions의 구성은 DUT의 입력과 출력에 관계된 properties를 정의하고, 입력 신호들의randomize 값 생성을 위해 data type 앞에 rand를 선언하고, 무의미한 randomize 값 생성을 막기 위해서 constraints 구문을 이용하여, 특정의 의도된 transaction 생성을 가능하게 한다[12].

UVM에서는 TLM(Transaction Level Modeling)을 도입하여, 테스트벤치 top 내부의 transaction level이라고 불리는 UVM_TEST를 구성하는 components 간의 data communication에 사용되는 data를 transaction이라고 한다[13].

sequence item class (eq_packet) 기술 시 모든 random 변수에 대한 constraints 구문 정의도 가능하지만, 관리와 다양한 testcases 생성 측면에 있어서 복잡도가 증가하는 단점도 따른다.

본 논문에서는 2개의 testcases를 생성하여 DUT의 검증을 하고자 하기에, 그림 5와 같이 sequence item class (eq_packet)에서는 common한 properties 정의만 하고, 각 test의 sequence내에서 constraints 구문을 정의하고자 한다.

4-by-4시스톨릭 어레이 구조의 transaction 정의에 앞서서 구조를 구성하는 각 PE의 x절편값과 y절편값의 초기값은 DUT 내부에서 0으로 초기화 하되, 예외적으로 P_1,P_2, P_3, P_4의 PE 를 기준으로 왼쪽의 상삼각 행렬를 구성하는Vector mode CORDIC PE인 R_{11},R_{21}, R_{31}, R_{41}의x 절편값은 1(망각계수) 로 초기화 하며, P_1,P_2, P_3, P_4의 PE 를 기준으로 오른쪽의 하삼각 행렬를 구성하는 R_{11}^{-1},R_{22}^{-1}, R_{33}^{-1}, R_{44}^{-1} 의 x절편값은 R_{11},R_{21}, R_{31}, R_{41}의 초기값의 inverse 값으로 초기화 한다.

Fig. 5의 transaction properties는 4-by-4 시스톨릭 어레이 구조의 첫번째 행의 R_{11}, R_{12}, R_{13}, R_{14}, P_1의 y절편 입력과 관계된 변수를 random 생성을 위해 rand로 선언하고, 등화기의 필터계수인 w(n) 출력을 나타내는 Weight_val_1, Weight_val_2, Weight_val_3, Weight_val_4 변수를 선언한다.

```
class eq_packet extends uvm_sequence_item;

  // Fields definition for randomization
  rand int R11_y_in;
    ...
  // Control fields definition
  bit signed [15:0] Weight_val_1;
    ...

  // Using Object utility macro, UVM Factory registry
  `uvm_object_utils_begin(eq_packet)
    // Using Field marco, each field operation (copy, clone, print...) enable definition
    `uvm_field_int(R11_y_in, UVM_ALL_ON)
      ...
  `uvm_object_utils_end
  // Object constructor - new() constructor definition
  function new (string name = "eq_packet");
      ...
endclass : eq_packet
```

Fig. 5. Properties define in sequence item class.

4.3 SEQUENCE

본 논문에서는 DUT의 검증을 위해 2개의 stimulus를 생성하여 test를 진행하기로 한다.

2개의 stimulus 생성에 차이점을 두기 위해서첫 번째 stimulus는 constraints random으로 생성하는 것을 보이고, 두 번째 stimulus는 정해진 입력 값을 설정하는 효과를 보이는 constraints stimulus 생성을 한다. UVM의 가장 큰 특징은 DUT로의 입력 stimulus에 대해서 random과 directed 생성 모두가 가능하다는 것이다.

sequence는 transaction 생성에 대한 process가구현된 UVM component가 아닌 object이다. 시나리오에 따라 다양한 transaction 생성을 위해 여러 sequence 작성이 가능하다[13].

Fig. 6과 같이 첫 번째 stimulus 생성을 위해 입력 값의 하나인 A(n) 을 구성하는 값들은 -20부터 20 사이의 정수만 사용하고, d(n)는 -30부터 160 사이의 정수만 사용하도록 constraints 제약조건을 설정해 random 생성한다.

```
class eq_random_1_seq extends uvm_sequence #(eq_packet);

  // Using Object utility macro, UVM Factory registry
    ...
  // Object Constructor - new() constructor definition
    ...
  // Repeat number
    rand int num;
  // Default Contraints for repeat number definition
  constraint repeat_num {num == 4;}

  // Specify body() task with start_item() and randomize() and finish_item()
  virtual task body();
    // Create instance
       ...
    // Process for random packets generation (repeat(num) begin ~ end)
    repeat(num) begin
      // Synchronize with sequencer and pre_do operation
      start_item(req);
      // Randomization with constraints
      assert(req.randomize() with{R11_y_in > -20; R11_y_in < 20; ... });
      // mid_do and post synchronization/body excution and post do operation
      finish_item(req);
    end
  ...
endclass : eq_random_1_seq
```

Fig. 6. 1st Sequence SV Code.

Fig. 7과 같이 두 번째 stimulus 생성을 위해 입력 값인 A(n) , d(n) 을 구성하는 값들은 특정 정수 값으로 사용하도록 constraints 제약조건을 설정해 생성한다.

```
class eq_base_seq extends uvm_sequence #(eq_packet);

  // Using Object utility macro, UVM Factory registry
    ...
  // Object Constructor - new() constructor definition
    ...
  // Objection handling before in eq_random_2_seq body() execution
  task pre_body();
    if (starting_phase != null)
       starting_phase.raise_objection(this, get_type_name());
    ...
  // Objection handling after in eq_random_2_seq body() body() execution
  task post_body();
    if (starting_phase != null)
       starting_phase.drop_objection(this, get_type_name());
    ...
endclass : eq_base_seq
```

```
class eq_random_2_seq extends eq_base_seq;

  // Using Object utility macro, UVM Factory registry
    ...
  // Object Constructor - new() constructor definition
    ...
  // Specify body() task with `uvm_do_with macros
  virtual task body();
    // Fields random packets generation with constraints
    `uvm_do_with(req, {R11_y_in == 2;
                       R12_y_in == 0;
                       R13_y_in == 0;
                       R14_y_in == 0;
                       P1_y_in  == 5; })
    ...

endclass : eq_random_2_seq
```

Fig. 7. 2nd Sequence SV Code.

4.4 SEQUENCER

Fig. 4와 같이 sequencer는 sequence 와 driver 사이에 위치해 있으며, driver의 transaction 전송 요청에 의해 sequencer에게 transaction 생성을 명령한다. sequence로부터 생성된 transaction을 수신하여 driver에게 전송하거나 driver로부터 response를 수신하여 sequencer로 전달하는 component이다[13].

Fig. 8과 같이sequencer는 sequence와 driver 사이의 communication을 위한 process는 상위 base class인 uvm_sequencer, uvm_sequencer_param_base, uvm_sequencer_base class에서 복잡한 처리를 담당하고 있기 때문에 상대적으로 uvm sequencer class code는 간단하다.

```
class eq_sequencer extends uvm_sequencer #(eq_packet);

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
endclass : eq_sequencer
```

Fig. 8. Sequencer SV Code.

4.5 DRIVER

driver는 sequencer 로부터 전달받은 transaction을 virtual interface를 이용하여 signal level로 변환한 후 DUT로 전달하거나, DUT로부터 전달받은 response를 sequencer로 전달하는 component이다[13]. Fig. 9와 같이driver class 정의 시 transaction type (eq_packet)의 parameter를 선언한다. 다음으로 DUT로 전달할 transaction을 signal level로 변환하기 위해 사용되는 virtual interface (eq_if)에 대한 handle (vif)을 선언한다.

```
class eq_driver extends uvm_driver #(eq_packet);

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  // Declare virtual interface handle
    ...
  // Retrieve the virtual interface handle that was set in config_db from testbench_top
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if ( !uvm_config_db#(virtual eq_if)::get(this, "", "vif", vif)
            ...
  // Run phase method
    virtual task run_phase(uvm_phase phase);
        ...
          get_and_drive();
          reset_signals(); //Set signal reset values
        ...
    task get_and_drive();
      ...
        forever begin
          seq_item_port.get_next_item(req); // Get random packets from eq_sequencer
          send_to_dut(req); // Drive data from received random packets into the DUT interface
          seq_item_port.item_done(); // Noty packets transfer completion to eq_sequencer
        end
      ...
  //Set signal reset default values
    task reset_signals();
        vif.r11_y_i         <= 32'b0;
        ...
  //Drive random packets into the DUT interface
    task send_to_dut(eq_packet packet);
        vif.r11_y_i         <= packet.R11_y_in;
        ...
endclass : eq_driver
```

Fig. 9. Driver SV Code.

class내의 build_phase()에서는 테스트벤치 top에서 uvm_config_db를 이용해 configuration database에서 set한 virtual interface에 대해서 실제 interface handle을 get 하는 code가 기술되어 있다.

run_phase()에서는 4.3절에서 작성한 2개의sequence에서 생성한transaction (eq_packet)을 sequencer와 driver간의 TLM port를 통해 가져오고, virtual interface handle를 통해서transaction 을 signal level로 변환한 후 DUT로 전송하는 code를 구현한다.

4.6 MONITOR

monitor는 DUT로부터의 response를 입력으로받아 signal level을 transaction level로 변환한 후 특정 TLM 형태인 analysis port를 통해 scoreboard와 같은 component로 drive 하는 component이다[13].

```
class eq_monitor extends uvm_monitor;

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  // Declare virtual interface handle
    ...
  // Declare analysis ports
    ...
  // Retrieve the virtual interface handle that was set in config_db from testbench_top
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if ( !uvm_config_db#(virtual eq_if)::get(this, "", "vif", vif)
            ...
  // Run phase method
    task run_phase(uvm_phase phase);
        ...
        collect_packet();
        ...
  // Collect signals from DUT interface and Assign to packet class fields
    task collect_packet();
        ...
         if( vif.resetn == 1 && vif.valid == 1 ) begin
           packet_collected.weight_val_1  <= vif.weight_val_1_o;
           ...
           item_collected_port.write(packet_collected);
        ...
endclass : eq_monitor
```

Fig. 10. MONITOR SV Code.

Fig. 10과 같이 monitor에서는 DUT로부터 수집한 DUT signal이 변환된transaction을 scoreboard로 전달하기 위한 TLM analysis port를 선언하고, build_phase()에서는 테스트벤치 top에서 uvm_config_db 를 이용해 configuration database에서 set 한 virtual interface에 대해서 실제 interface handle을 get 하는 code가 기술되어 있다. run_phase()에서는 DUT로의 최종 결과물인 등화기 필터계수를 수집하여 virtual interface를 이용하여 signal level를 transaction level로 변환하고 scoreboard로 해당 transaction을 write하는code가 구현되어 있다.

4.7 AGENT

agent는 sequencer, driver, monitor와 같은 UVM componets를 그룹화 하는 component이고, DUT와 interface를 통해 통신한다[13].

```
class eq_agent extends uvm_agent;

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  // Create handles to all agent components: monitor,driver,sequencer
    eq_monitor monitor;
    ...
  //Set Agent property ("UVM_ACTIVE" or "UVM_PASSIVE")
    ...
  // build phase method
    virtual function void build_phase(uvm_phase phase);
        ...
      // Both ACTIVE and PASSVIE agents need a monitor
        monitor = eq_monitor::type_id::create("monitor", this);
      // If this agent is ACTIVE, need a both driver instance and sequencer instance
        if(is_active == UVM_ACTIVE) begin
            driver = eq_driver::type_id::create("driver", this);
            ...
  // Connect the sequencer port and driver port inside the agent
    virtual function void connect_phase(uvm_phase phase);
        ...
        driver.seq_item_port.connect(sequencer.seq_item_export);
        ...
endclass : eq_agent
```

Fig. 11. AGENT SV Code.

Fig. 11과 같이 monitor, driver, sequencer를instance 하여 그룹화 시키고, active agent (sequencer, driver 포함)를 만들기 위하여 is_active를 UVM_ACTIVE로 설정한다. connect_phase()에서는 transaction 통신을 위해sequencer와 driver간의 TLM port를 연결한다.

4.8 ENV

agent와 scoreboard와 같은 components를 그룹화 한 container component이다. 하드웨어 spec에 따라서 multiple sub agent 또는 multiple sub env를 포함하는 구성이 가능하고 scoreboard추가 구성도 가능하다[13].

Fig. 4에서와 같이 hierarchy 상 uvm_agent와 uvm_scoreboard를 포함하고 있으며, Fig. 12는 위의 관계에 대해서 code로 명시하고 있다. agent와 scoreboard의 handle을 선언하고 있고, hierarchy 상 env 내부 agent 아래의 monitor에서 선언한 TLM analysis port와 env 아래의 scoreboard에서 선언한 TLM analysis imp port에 연결하기 위해 connect_phase()에서 기술하고 있다.

```
class eq_env extends uvm_env;

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  // Create handles to all env components: agent, scoreboard
    eq_agent agent;
    eq_scoreboard sb;
  // build phase method
    virtual function void build_phase(uvm_phase phase);
      // Instanciate  components ( agents and scoreboard)
      agent = eq_agent::type_id::create("agent", this);
      ...
    virtual function void connect_phase( uvm_phase phase );
      // Connect  analysis port from monitor to the scoreboard
      agent.monitor.item_collected_port.connect(sb.item_collected_export_eq);
      ...
endclass : eq_env
```

Fig. 12. ENV SV Code.

4.9 SCOREBOARD

scoreboard는 monitor로부터 sampling 한 DUT의 Actual data와 Reference model의 Expected data를 compare하여 DUT의 function을 check 하는 component이다[13].

Fig. 4에서와 같이hierarchy 상 env 아래에 위치해 있다.

```
class eq_scoreboard extends uvm_scoreboard;

  // Declare an "eq_que" queue to store the received eq_packets from monitor
   ...
  // Declare an "eq_mem[4]" array to store only the required data from eq_que
   ...
  // Declare an "exp_mem[4]" array to store data from matlab results file
   ...
  // Declare analysis impl ports to receive packets from monitor
    uvm_analysis_imp_eq #(eq_packet, eq_scoreboard) item_collected_export_eq;
  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  // Build phase & Instanciate  analysis_impl_port
  virtual function void build_phase( uvm_phase phase );
    item_collected_export_eq = new( "item_collected_export_eq", this );
    ...
  //Recives the packets from monitor and pushes into queue
  virtual fuction void write_eq (eq_packet eq_pkt);
    eq_que.push_back(eq_pkt);
     ...
```

Fig. 13. SCOREBOARD SV Code (1 of 2).

Fig. 13은 2 개의 test를 위한 MATLAB을 이용한 Reference data를 저장하고 있는 파일을 include 하고 있다. class 내부는 monitor가 sampling 한transaction(eq_packet) 중에서 등화기 필터계수 4개의 property만을 저장하기 위해 배열 변수 (eq_mem)를 선언하였다. 또한, Reference data를 저장하고 있는 파일에서 값을 읽고 저장하기 위해 배열 변수(exp_mem)을 선언한다.

monitor의 TLM analysis port와 연결하기 위해 scoreboard에서는 TLM analysis imp port를 선언해 준다. monitor에서 사용한 write 함수에 대한 실제 transaction(eq_packet)을 queue에 write 하는 code를 code를 구현한다.

```
virtual task run_phase (uvm_phase phase);
  // Read datas from MATLAB results file
  `ifdef RANDOM_1
    fd = $fopen ("random_1_test.txt" , "r");
    ...
  // Assign "fd" file contexts to exp_mem array
  do begin
    fd1 = $fscanf(fd, "%f", exp_mem[i]);
    ...
  // Compare's the only the required actual data (Weight_val_1..) from eq_que (stored overall packet fields)
  // and the referecen matlab data from exp_mem
  forever begin
    wait (eq_que.size() > 0);
    eq_item = eq_que.pop_front();
    eq_mem[0] = eq_item.weight_val_1;
    ...
    for (int j=0 ; j < $size(eq_mem) ; j = j+1) begin
      if (exp_mem[j] - $itor(eq_mem[j]*SF) <= 0.2) begin
        // PASS statement
          ...
      end
      else begin
        // FAIL statement

endclass : eq_scoreboard
```

Fig. 14. SCOREBOARD SV Code (2 of 2).

Fig. 14는 Reference data가 저장되어 있는 파일을 읽어 exp_mem에 저장하고, transaction이 저장되어 있는 queue에서 등화기 필터 계수 4개의 property만을 저장하기 위해 배열 변수 (eq_mem)에 assign 한다. 마지막으로 exp_mem과 eq_mem의 값을 비교하여 PASS, FAIL을 판단하는 checker기능을 하는 code를 구현한다.

4.10 TEST

UVM components의 top level class이며 env를 포함하고 있으며, 특정sequence를 선택하여 다양한 testcases를 생성하여 실행할 수 있는 component이다[13].

Fig. 15는 eq_random_1_seq를 가지고 test class를 작성한 code이다. run_phase() 내에서 eq_random_1_seq를 randomize 하고 sequence를 실행시키는 sequencer와 mapping 함과 동시에start함수를 이용해 sequence를 실행한다.

```
class eq_random_1_test extends uvm_test;

  // Using Component utility macro, UVM Factory registry
    ...
  // Create handles to env components
  eq_env  env;
  // Component Constructor - new() constructor definition
    ...

  virtual function void build_phase( uvm_phase phase );
    // Instanciate  components (env)
    env = eq_env::type_id::create( "env", this );
    ...

  virtual task run_phase( uvm_phase phase );
    // Create eq_random_1_seq sequence
    seq_1 = eq_random_1_seq::type_id::create("seq_1");

    // Objections control
    phase.raise_objection(this, "seq_1");
    // To do repeat num randomize
    seq_1.randomize();
    // Sequence start
    seq_1.start(env.agent.sequencer);
    // Objections control
    phase.drop_objection(this, "seq_1");
    `uvm_info("eq_random_1_test", "Complete transactions ...", UVM_LOW)
    ...
endclass : eq_random_1_test
```

Fig. 15. TEST with 1st sequence (eq_random_1_seq) SV Code.

```
class eq_base_test extends uvm_test;

  // Using Component utility macro, UVM Factory registry
    ...
  // Create handles to env components
  eq_env  env;
  // Component Constructor - new() constructor definition
    ...
  // UVM build_phase() phase
  function void build_phase(uvm_phase phase);
     // Instanciate  components (env)
    env = eq_env::type_id::create("env", this);
    ...
  task run_phase(uvm_phase phase);
    // set drain time
    phase.phase_done.set_drain_time(this, 200ns);
    ...
endclass : eq_base_test

class eq_random_2_test extends eq_base_test;

  // Using Component utility macro, UVM Factory registry
    ...
  // Component Constructor - new() constructor definition
    ...
  function void build_phase(uvm_phase phase);
    // Set to the default sequence to execute
    uvm_config_wrapper::set(this, "env.sequencer.run_phase",
                            "default_sequence",
                            eq_random_2_seq::type_id::get());
    ...
endclass : eq_random_2_test
```

Fig. 16. TEST with 2nd sequence (eq_random_2_seq) SV Code.

Fig. 16은 eq_random_2_seq를 가지고 test class를 작성한code이다. Fig. 15와는 다른 방법으로 test class내의 build_phase()에서 실행할 sequence를 미리 설정할 수 있다. default sequence는 eq_random_2_seq으로 지정하고 sequencer의 run_phase에 연결함으로 sequence가 실행된다.

test class내에서 sequence를 실행하는 동일한 효과를 보이는 2 가지 방법을 보여 주기 위해 Fig. 15와 Fig. 16에서와 같이 다른 접근방법의 code를 작성하였다.

5. 실험결과

5.1 eq_random_1_test

eq_random_1_test 내부에서 eq_random_1_seq의 randomize 한 값들은 다음과 같다.

$$A(n)=\begin{bmatrix} 10 & -6 & 8 & 9 \\ -13 & -15 & 11 & -8 \\ 11 & -14 & -8 & -2 \\ 9 & 14 & -9 & 9 \end{bmatrix}, d(n)=\begin{bmatrix} 151 \\ 64 \\ -36 \\ -29 \end{bmatrix}$$

아래는 확장 QRD-RLS 등화기용 시스톨릭 어레이 구조의 각각의 clock에 따른 결과이다.

Table 2의 7 clock이 지나면 $R^{-1}(n)$ 과 $P(n)$ 을 얻을 수 있다. $P(n)$ 은 노랑색으로 표시된 P1, P2, P3, P4로 구성된 열 벡터 이고 $R^{-1}(n)$ 은 $P(n)$ 의 우측 4-by-4 하삼각행렬의 녹색으로 표시된 값을 transpose하여 도출된다.

Table 2. Computation result in each clock (eq_random_1_test)

(1, 10) (10.05, 0)	(0, -6) (-5.97, -0.6)	(0, 8) (7.96, 0.8)	(0, 9) (8.96, 0.9)	(0, 151) (150.02, 15.04)	(1, 0) (0.1, -1.0)				1 clock
	1	0	0	0	0	1			
		1	0	0	0	0	1		
			1	0	0	0	0	1	
(10.05, -13) (16.43, 0)	(-5.97, -15) (8.22, -13.90	(7.96, 11) (-3.83, 13.02	(8.96, -8) (11.81, 2.19)	(150.02, 64) (41.26, 158.0)	(0.1, 0) (0.06, 0.08)				2 clock
	(1, -0.6) (1.16, 0)	(0, 0.8) (-0.41, 0.68)	(0, 0.9) (-0.46, 0.77)	(0, 15.04) (-7.71, 12.91)	(0, -1) (0.51, -0.85)	(1, 0) (0.86, 0.51)			
		1	0	0	0	0	1		
			1	0	0	0	0	1	
(16.43, 11) (19.77, 0)	(8.22, -14) (-0.96, 16.20)	(-3.83, -8) (-7.63, -4.52)	(11.81, -2) (8.7, -8.23)	(41.26, -36) (14.27, -52.86)	(0.06, 0) (0.05, -0.03)				3 clock
	(1.16, -13.9) (13.94, 0)	(-0.41, 13.02) (-13.01, 0.68)	(-0.46, 2.19) (-2.22, -0.27)	(-7.71, 158) (-158.077, 5.52)	(0.51, 0.08) (-0.04, 0.51)	(0.86 ,0) (0.07, 0.86)			
		(1, 0.68) (1.21, 0)	(0, 0.77) (0.43, 0.64)	(0, 12.91) (7.29, 10.66)	(0, -0.85) (-0.48, -0.7)	(0, 0.51) (0.29, 0.42)	(1, 0) (0.83, -0.56)		
			1	0	0	0	0	1	
(19.77, 9) (21.72, 0)	(-0.96, 14) (4.93, 13.14)	(-7.63, -9) (-10.68, -5.03)	(8.7, 9) (11.64, 4.59)	(14.27, -29) (0.97, -32.3)	(0.05, 0) (0.05, -0.02)				4 clock
	(3.94, -16.2) (21.38, 0)	(-13.01, -4.52) (-5.07, -12.81)	(-2.22, -8.23) (4.79, -7.05)	(-158.08, -52.86) (-63.05. -154.29)	(-0.04, -0.03) (0, -0.05)	(0.07, 0) (0.05, 0.05)			
		(1.21, 0.68) (1.4, 0)	(0.43, -0.27) (0.24, -0.45)	(7.29, 5.52) (9.06, 1.24)	(-0.48, 0.52) (-0.17, -0.69)	(0.29, 0.86) (0.67, 0.6)	(0.83, 0) (0.72, -0.4)		
			(1, 0.64) (1.18, 0)	(0, 10.66) (5.71, 9.0)	(0, -0.7) (-0.38, -6.0)	(0, 0.42) (0.23, 0.36)	(0, -0.56) (-0.3, -0.48)	(1, 0) (0.84, -0.54)	
21.72	4.93	-10.68	11.64	0.97	0.05				5 clock
	(21.38, 13.14) (25.09, 0)	(-5.07, -5.03) (-6.95, -1.63)	(4.79, 4.59) (6.48, 1.4)	(-63.05, -32.3) (-70.63, 5.49)	(0, -0.02) (-0.01, -0.02)	(0.05, 0) (0.04, -0.02)			
		(1.39, -12.81) (12.82, 0)	(0.24, -7.05) (7.04, -5.02)	(9.06, -154.29) (154.35, -7.64)	(-0.17, -0.05) (0.03, -0.17)	(0.67, 0.05) (0.02, 0.67)	(0.72, 0) (0.08, 0.72)		
			(1.18, -0.45) (1.27, 0)	(5.71, 1.24) (4.89, 3.2)	(-0.38, 0.69) (-0.6, 0.51)	(0.23, 0.6) (0, 0.64)	(-0.3, -0.4) (-0.14, -0.49)	(0.84, 0) (0.79, 0.3)	
21.72	4.93	-10.68	11.64	0.97	0.05				6 clock
	25.09	-6.95	6.48	-70.63	-0.01	0.04			
		(12.82, -1.63) (12.98, 0)	(7.04, 1.4) (6.8, 2.28)	(154.35, 5.49) (152.43, 24.83)	(0.03, -0.02) (0.03, -0.01)	(0.02, -0.02) (0.02, -0.02)	(0.08, 0) (0.08, 0.01)		
			(1.27, -0.52) (1.37, 0)	(4.89, -7.64) (7.42, -5.22)	(-0.6, -0.17) (-0.49, -0.39)	(0, 0.67) (-0.26 0.62)	(-0.4, 0.01) (-0.2, 0.35)	(0.73, 0) (0.38, -0.63)	
21.72	4.93	-10.68	11.64	0.97 (P1)	0.05				7 clock
	25.09	-6.95	6.48	-70.63(P2)	-0.01	0.04			
		12.98	6.8	152.43(P3)	0.03	0.02	0.08		
			(1.37, 2.28) (2.66, 0)	(7.42 ,24.83) (25.1,(P4), 6.44)	(-049, -0.01) (-0.26, 0.41)	(-0.26, -0.02) (-0.15, 0.21)	(-014, 0.72) (-0.26, 0.61)	(079, 0) (0.73, 0.3)	

$$R^{-1}(n)=\begin{bmatrix}0.05 & -0.01 & 0.03 & -0.26\\ 0 & 0.04 & 0.02 & -0.15\\ 0 & 0 & 0.08 & -0.20\\ 0 & 0 & 0 & 0.38\end{bmatrix}, P(n)=\begin{bmatrix}0.97\\ -70.63\\ 152.43\\ 25.1\end{bmatrix}$$

위의 $R^{-1}(n)$ 과 $P(n)$ 을 가지고 식 (5)를 이용하여 등화기의 필터계수 $w(n)$ 이 계산된다.

$$w(n)=\begin{bmatrix}-1.1983\\ -3.5416\\ 7.1744\\ 9.538\end{bmatrix} \quad (11)$$

Fig. 17은 UVM 기반의 test를 수행하여 (11)에서 도출된 DUT의 Actual data와 MATLAB을 이용한 Reference data를 compare 한 simulation log결과를 보여주고 있다. scoreboard 내의 checker에서는 MATLAB Reference data와 DUT Actual data의 오차를 감안하여 오차범위를 <=0.2로 설정하고, 오차범위 이내에 값이면 PASS로 판정한다.

```
UVM_INFO eq_scoreboard.sv(60) @ 350 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:-1.19,Expected data:-1.05
UVM_INFO eq_scoreboard.sv(60) @ 370 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:-3.54,Expected data:-3.39
UVM_INFO eq_scoreboard.sv(60) @ 390 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:7.17,Expected data:7.32
UVM_INFO eq_scoreboard.sv(60) @ 410 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:9.53,Expected data:9.69
```

Fig. 17. Check Result the Actual data and the Expected data on the SCOREBOARD (eq_random_1_test).

5.2 eq_random_2_test

eq_random_2_test 내부에서 eq_random_2_seq의 randomize 한 값들은 다음과 같다.

$$A(n)=\begin{bmatrix}3 & 0 & 0 & 0\\ 11 & 3 & 0 & 0\\ -1 & 11 & 3 & 0\\ 4 & -1 & 11 & 3\end{bmatrix}, d(n)=\begin{bmatrix}5\\ 18\\ -12\\ 8\end{bmatrix}$$

아래는 확장 QRD-RLS 등화기용 시스톨릭 어레이 구조의 각각의 clock에 따른 결과이다.

Table 3. Computation result in each clock (eq_random_2_test)

(0.01, 3) (3, 0)	0	0	0	(0, 5) (5, 0.01)	(100, 0) (0.37-100)			1 clock
	0.01	0	0	0	0	100		
		0.01	0	0	0	0	100	
			0.01	0	0	0	0	100

(3,11) (11.25,0)	(0,3) (2.59,0.63)	0	0	(5, 18) (18.67,-0.59)	(0.37,0) (0.08,-0.36)		2 clock
	0.01	0	0	(0, 0.01)	(0,-100)	(100,0) (1.58,-99.99)	
		0.01	0	0	0	0	100
			0.01	0	0	0	0	100

(11.25,-1) (11.29,0)	(2.59, 11) (1.67, 11.10)	(0,3) (-0.22, 2.65)	0	(18.67,-12) (19.59,-10.41)	(0.08,0) (0.08,0.00)			3 clock
	(0.01, 0.63) (0.63,0)	0	0	(0,-0.59) (-0.59,-0.01)	(0,-0.36) (-0.36,-0.01)	(1.58, 0) (0.09, -1.58)		
		0.01	0	(0, 0.01)	(0, -100)	(0, -99.99)	(100,0) (6.61, -99.78)	
			0.01	0	0	0	0	100

(11.29, 4) (11.91, 0)	(1.67, -1) (1.30, -1.42)	(-0.22, 11) (3.28, 10.43)	(0, 3) (0.85, 2.53)	(19.59, 8) (21.13, 1.33)	(0.09, 0) (0.08, -0.03)			4 clock
	(0.63, 11.10) (11.13,0)	(0, 2.65) (2.65, 0.15)	0	(-0.59, -10.41)) (-10.43. -0.00)	(-0.36, 0.00) (-0.01, 0.36)	(0.09, 0) (0.09, 0.01)		
		0.01	0	(0, -0.01)	(0, -0.01)	(0, -1.58) (-1.58, -0.10)	(6.61, 0) (0.09, -6.61)	
			0.01	(0, 0.01)	(0, -100)	(0, -99.99)	(0, -99.78)	(100,0) (27.12,-96.25)

11.91	1.30	3.28	0.85	21.13	0.08			5 clock
	(11.13,-1.42) (11.22.0)	(2.65, 10.43) (1.31, 10.68)	(0, 2.53) (-0.32, 2.51)	(-10.43, 1.33) (-10.52, 0.00)	(-0.01, -0.03) (-0.01, -0.03)	(0.09,0) (0.08,-0.03)		
		(0.01, 0.15) (0.15,0)	0	(0,-0.00) (-0.00, 0.00)	(0, 0.36) (0.36, 0.02)	(-1.58, 0.01) (-0.01, 1.58)	(0.09, 0) (0.09, 0.00)	
			0.01	(0, -0.01)	(0, -0.01)	(0, -010)	(0,-6.61) (-6.36,-1.79)	(27.12,0) (0.10,-27.12)

21.72	4.93	-10.68	11.64	0.97	0.05			6 clock
	11.22	-1.31	-0.32	-10.52	-0.01	0.08		
		(0.15, 10.68) (10.69, 0.00)	(0, 2.51) (2.51, 0.04)	(-0.00, 0.00) (0.00, 0.00)	(0.36, -0.03) (-0.02, -0.36)	(-0.01, -0.03) (-0.01, -0.03)	(0.09, 0) (0.08, -0.04)	
			(0.01, 0.00)	(0.00, 0.02)	(0.00, 0.02)	(0.00, 1.58) (1.52, 0.43)	(-6.36, 0.00) (-0.02, 6.36)	(0.10,0) (0.10, 0.02)

21.72	4.93	-10.68	11.64	0.97(P1)	0.05			7 clock
	11.22	-1.31	-0.32	-10.52(P2)	-0.01	0.08		
		10.69	2.51	0.00(P3)	-0.02	-0.01	0.08	
			(0.01, 0.04) (0.04, 0)	(0.00, 0.00) (0.00(P4), 0.00)	(0.00, -0.36) (-0.35, -0.10)	(1.52, -0.03) (-0.02, -1.52)	(-0.02, -0.04) (-0.01, -0.05)	(0.10,0) (0.10, 0.02)

Table 3의 7 clock이 지나면 $R^{-1}(n)$ 과 $P(n)$ 을 얻을 수 있다. $P(n)$ 은 노랑색으로 표시된 P1, P2, P3, P4로 구성된 열 벡터 이고 $R^{-1}(n)$ 은 $P(n)$ 의 우측 4-by-4 하삼각행렬의 녹색으로 표시된 값을 transpose하여 도출된다.

$$R^{-1}(n)=\begin{bmatrix}0.05 & -0.01 & -0.02 & -0.35\\ 0 & 0.08 & -0.01 & -0.02\\ 0 & 0 & 0.08 & -0.01\\ 0 & 0 & 0 & 0.10\end{bmatrix}, P(n)=\begin{bmatrix}0.97\\ -10.52\\ 0.00\\ 0.00\end{bmatrix}$$

위의 $R^{-1}(n)$ 과 $P(n)$ 을 가지고 식 (5)를 이용하여 등화기의 필터계수 $w(n)$ 이 계산된다.

$$w(n)=\begin{bmatrix}0.1537\\ -0.8416\\ 0\\ 0\end{bmatrix} \quad (12)$$

Fig. 18은 UVM 기반의 test를 수행하여 (12)에서 도출된 DUT의 Actual data와 MATLAB을 이용한 Reference data를 compare 한 simulation log결과를 보여주고 있다.

```
UVM_INFO eq_scoreboard.sv(60) @ 350 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:0.15,Expected data:0.32
UVM_INFO eq_scoreboard.sv(60) @ 370 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:-0.84,Expected data:-0.66
UVM_INFO eq_scoreboard.sv(60) @ 390 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:0.00,Expected data:0.19
UVM_INFO eq_scoreboard.sv(60) @ 410 ns: uvm_test_top.env.sb [eq_scoreboard]
COMPARE PASS: Actual data:0.00,Expected data:0.16
```

Fig. 18. Check Result the Actual data and the Expected data on the SCOREBOARD (eq_random_2_test).

6. 결 론

본 논문에서는 확장 QRD-RLS 등화기용 시스톨릭 어레이 구조의 검증에 대해서 표준화한 신뢰성이 높은 UVM 기반의 Class library를 활용하여 테스트벤치를 구축하는 방법을 제안하였다.

constraints randomize을 통해 다양한 시나리오 stimulus를 생성하는 것을 확인할 수 있었다. 또한 simulation을 통해 실제 DUT 출력과 MATLAB Reference 출력을 비교하는 test를 진행하여 UVM 테스트벤치의 재사용성과 효용성을 확인할 수 있었다. 추후에 다양한 입력 신호의 길이를 변화시켜 등화기의 성능을 비교하는 것에 적용이 가능하리라 예상된다.

이것은 기존 Verilog 기반의 테스트벤치 검증방식의 재사용성과 확장성, 유연성을 보장하기 어렵다는 단점을 획기적으로 개선하여 기존에 구축된 검증환경과 VIP를 재사용함으로써 검증환경의 개발 시간을 단축시키는 장점이 있다. 또한 테스트벤치의 유지,관리가 용이하여 다양한 프로젝트에 적용이 가능하다는 장점이 있다.

앞으로는 반도체의 복잡도가 더욱 증가하여 검증이 더욱 중요하리라 예상된다. 이에 검증 프로세스의 표준화 , 공동 업무화를 위해서는 UVM 기반의 검증이 보편화가 되리라 예상된다.

향후, MATLAB을 이용한 Reference data가 아닌 C를 이용한 DPI-C의 추가적인 개발도 필요할 것으로 판단된다.

감사의 글

다음의 성과는 과학기술정보통신부와 연구개발특구진흥재단이 지원하는 과학벨트지원사업으로 수행된 연구결과입니다.

참고문헌

1. Yong-Sung Kim, Yong-Seok Lim and Dae-Ki Hong, “Low Power SoC Modem Design for High-Speed Wireless Communications”, Journal of the Semiconductor & Display Technology, Vol. 9, No. 2, pp. 7~10, June 2010
2. Jang-Youn Lee, Jin-Woong Cho and Dae-Ki Hong,” Improved Binary CDMA Modem Design for High-Speed Wireless Communications”, Journal of the Semiconductor & Display Technology Vol. 9, No. 2, pp. 11~14, June 2010
3. S. Haykin, "Adaptive filtering theory," 3rd Ed. Prentice Hall, New Jersey.
4. S. Haykin, A. H. Sayed, J. R. Zeidler, P. Yee, and P. C. Wei, "Adaptive tracking of linear time-variant systems by extended RLS algorithms," IEEE Trans, Signal Proc. Vol. 45, No. 5, pp. 1118-1127, May 1997.
5. L. Gao and K. K. Parhi, "Hierarchical pipelining and folding of QRD-RLS adaptive filters and its application to digital beamforming," IEEE Trans, Circuits and Systems II, Vol. 47, No. 12, pp. 1503-1519, Dec. 2000.
6. Z. Chi, J. Ma and K. K. Parhi, "Hybrid annihilation transformation for pipelining QRD- based least square adaptive filters," IEEE Trans, Circuits and Systems II, Vol. 48, No. 7, pp. 661-674, Jul. 2001.
7. Sangun Jo, Jonghwan Lee and Yongwoo Kim,“A Design and Implementation of 32-bit Five-Stage RISC-V Processor Using FPGA”, Journal of the Semiconductor & Display Technology, Vol. 21, No. 4, pp.27~32, December 2022
8. J. E. Volder, "The CORDIC trigonometric computing technique," IRE Trans, on Electronic Computer 8, pp. 330-334, 1959.
9. Y. H. Hu, "CORDIC-based VLSI architecture for digital signal processing," IEEE Signal Processing Magazine, Vol. 9, pp. 16-35, 1992.
10. T. Z. Mingqian, A. S. Madhukumar and F. Chin, "QRD-RLS adaptive equalizer and its CORDIC- based implementation for CDMA systems", International Journal on Wireless & Optical Communications, Vol. 1, No. 1, pp. 25-39, 2003.
11. K. I. Lee, J. H. Lee, J. K. Lee and Y. S. Cho, "A compact CORDIC algorithm for synchronization of carrier frequency offset in OFDM modems," IEICE Transactions on Communications, Vol. E89-B, No. 3, pp. 952-954, 2006.
12. IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language.
13. Universal Verification Methodology (UVM) 1.2 User’s Guide, Accellera, October 8, 2015.

접수일: 2024년 1월 11일, 심사일: 2024년 3월 6일,
게재확정일: 2024년 3월 20일