특 집

# 통합된 검증 방법론
# (UVM ; Unified Verification Methodology)

김 규 흥
케이던스 코리아

## I. 서 론

설계자는 좀 더 적은 노력 및 투자로 좀 더 짧은 시간에 설계한 칩을 검증하고자 희망하며, EDA 업체들도 툴로 검증문제을 해결하기 위해서 많은 시간을 노력하고 있다. 또한 모든 검증 문제는 Speed와 Efficiency를 높이는데 있다. SoC 검증팀은 SoC의 초기 구조 확정, 시스템 요소의 통합, 시스템 검증 부분에 책임을 져야 할 경우가 있다. 그런데, 지금까지의 SoC 설계는 일반적으로 독립된 팀들에 의해 이루어지고, 각 팀은 그들만의 독립된 검증 기법을 사용해 왔다. SoC가 서로 독립되어 있지만 협력관계인 IP 팀, 내부 핵심 팀, 알고리즘 개발 팀, Analog/RF 팀 그리고 전단계 팀에서 얻어진 결과들로 이루어질 경우, 시스템 요소의 통합이 쉽지 않다. 또한, SoC 검증팀이 시스템 요소의 통합을 테스트하기 위해서는 전혀 새로운 검증 환경을 또 다시 만들어야 한다. SoC 설계에서 매우 중요해지는 Embedded Software팀은 시스템 요소의 통합이 완성될 때까지 기다려야 하기 때문에 시스템 검증 및 시스템 실제 구현을 시작하기 까지는 많은 시간이 필요하다.

지금까지 나열된 문제점들의 원인검증에서 Speed와 Efficiency를 증대시키는 것을 방해하는 것이 뭔가를 분석한 결과 근본 원인이 조각 조각 나누어진 설계 환경이라는 것을 알았다. 즉 검증 과정에서 각 단계별로 구분되고(시스템 레벨과 구체화된 아키텍쳐 레벨), 서로 다른 도메인(디지털과 아날로그, 하드웨어와 소프트웨어), 유사하지만 다른 프로젝트등은 설계시에 사용하는 서로다른 호환성 없는 언어와 툴, 기술에서 기인된다고 보여진다. 이런 문제를 해결하기 위한 유일한 길은 통합된 방법론으로 전체 검증 과정을 단일화하는 것이다.

통합검증이란 하나의 디자인, 시뮬레이션 환경에서 한 부분도 빠짐없는 전체 시스템이 시뮬레이션 스피드의 부담없이 돌아가는 것으로, 많은 시간을 검증이 아닌 설계과정에 투자할 수 있으며 그러므로 보다 완성도 높은 디자인을 더 빠른 시간에 시장에 내어놓을 수 있을것이다. 본고에서는 통합 검증방법론의 전체적인 흐름 및 툴의 구조에 대하여 기술하였다.

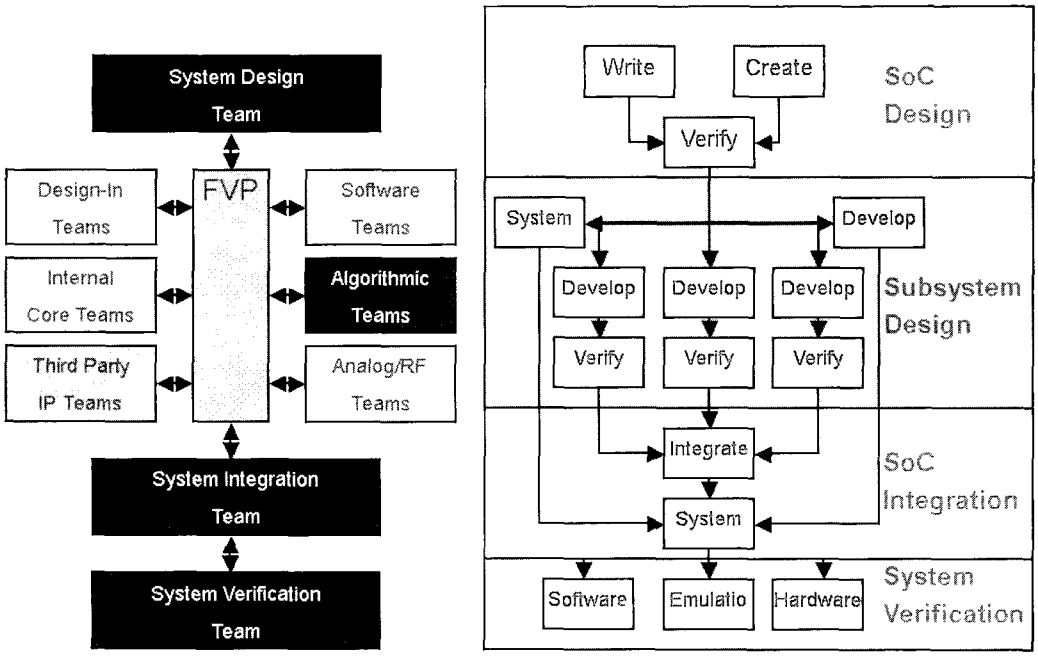
## II. 통합 검증 방법론

그러면 SoC 설계 및 검증을 위해서는 기존의 설계 방법 및 환경을 어떻게 바꾸어야 하나? 여기서 제안하는 통합 검증 방법론은 그 물음에 대한 하나의 대안으로 충분하리라 생각된다.

통합된 검증 방법론은 특정 EDA 업체의 Solution에 국한되지 않는 검증을 위한 청사진으로써 방법론을 말하고 있다.

따라서 설계 팀들은 이 방법론을 채택하여 오늘날 그 들에게 가장 의미 있는 부분, 실현 가능한 부분에 도입할 수 있고, 향후의 작업을 위한 최상의 실천적인 경험에 근거한 방법론으로 사용할 수 있다.

우리는 전체 방방론을 모든 디자인에 적용하라




〈그림 1〉 Unified Verification Methodology

고 주장하지는 않는다. 그러나 방법론의 일부는 오늘날 모든 설계 팀에게서 유용하게 사용할 수 있다.

Speed와 Efficiency에 증대에 중점을 두어 정립된 "통합된 검증 방법론"의 근간을 이루는 몇 가지 기본 원칙은 아래와 같다.

첫째 검증에 있어서 난처한 처지에 빠지게 하는 실체는 위험 관리이다. 디자인이 결함이 없다고 100% 확신하는 것은 불가능하다. 그래서 우리는 설계가 제품화된 후에 결함이 발견될 위험과 설계 중에 결함을 찾아 해결하고자 하는 시간과 노력·투자 사이에서 손익을 비교하여 고려해야 한다.

둘째는 검증은 미리 알아서 능동적으로 대처하는 것이다. 오늘날 검증과 관련하여서 많은 장애물과 종속물들이 있다. 우리는 설계를 완성해 나가기 위해서 이런 장애물과 종속물들을 제거하는 방법들을 찾을 필요가 있다.

셋째는 검증의 요령은 작업량(Workload)와 효과(Payback)의 손익을 고려하여 적절한 시점에 적절한 툴을 사용하는 것이다. 이는 설계의 결함을 발견하기 위한 검증에는 많은 툴과 기법이 있고, 각각은 검증과정 과정에서 각자 적당한 자리를 차지하고 있기 때문이다.

마지막으로 검증은 사고 방식에 있다. 설계자가 하드웨어를 생각하야 하듯이, 검증 엔지니어는 결함을 생각해야 한다. 검증 엔지니어는 어떻게 설계가 동작하게 되는지를 생각해야 할 뿐만 아니라, 어떻게 하면 동작하지 않을 수 있는지와 결함이 있는 것 또는 없는 것을 검증하는 방법까지 생각해야 한다.

통합된 검증 방법론은 모든 격리된 팀을 공통의 방법론으로 하나로 묶어 준다.

이것은 여러 팀들이 같은 툴을 같은 방법으로 사용한다는 의미는 아니라, 팀들이 일을 하는데 사용하는 툴 및 과정이 격리되는 대신에 함께 하나로 묶이는 것을 의미한다.

통합된 검증 방법론(UVM ; Unified Verification Methodology)의 핵심에는 FVP(Functional Virtual Prototype)라는 공통의 설계 모델이 있다. 모든 설계 영역 및 검증 단계 사이에서 공통 모델을 위한 원천을 제공하는 FVP에

의해서 하나로 묶여진다.

모든 설계 영역 및 검증 단계 사이에 공통 모델을 제공하는 FVP는 Embedded Software 팀이 설계 초기서부터 작업을 시작하게 해주며, 시스템 요소의 통합(System Integration)이 더 이상 악몽이 아니게 해주며, 시스템 검증 및 구현을 전체 설계 과정에서 훨씬 초기부터 시작하게 해준다.

흔히들 혼돈하는 용어의 정의를 간단히 하고자 한다.

방법론(Methodology)와 흐름(flow)의 차이는 살펴보면, 방법(Method)은 개별 작업(individual task)을 성취하는 위해서 어떻게 해야 한 것이고, 방법론(Methodology)은 한끼의 식사를 차리는 것처럼 활동(Activity)을 수행하기 위해 사용되는 방법의 집합체이고, 흐름(flow)은 통상 한 반찬의 요리법처럼 한 방면에서의 연속적인 흐름으로 정의한다. 예를 들면, Verification Tools flow는 특정 설계(프로젝트의 디지털 구현 측면)를 시작부터 끝까지 어떻게 검증할까를 단계 단계별로 기술한 것이다.

## III. 모델링 수준(Abstraction Level)

통합된 검증 방법론(UVM ; Unified Verification Methodology)의 핵심인 FVP(Functional Virtual Prototype)로 논의를 진행하기에 앞서 추상화 수준에 대한 간단한 정리가 필요할 것같아 참고로 기술하고자 한다.

시스템을 구성하는 모든 모듈들의 설계가 항상 같은 추상화 수준을 유지하며 진행되는 경우는 극히 드물며, 이러한 방법론의 구분은 별 의미가 없고 서로 반복적인 과정이라는 것을 인식할 필요가 있을 것이다. 실제로 시스템 설계를 하다 보면 처음부터 명확한 추상화 수준을 가지고 시작되는 경우는 찾아보기 힘든다고들 말한다. 시스템 설계 과정에서 새로운 모델을 추가할 때, 이미 작성된 하드웨어나 소프트웨어 모델을 다양한 수준에서 다시 모델링하고 테스트 벤치를 작성하는 따위의 일은 아주 일반적이다. 결국 서로 다른 추상화 수준의 모델들이 복합적으로 사용되어 설계가 진행되는 것은 아주 흔한 일인 것이다.

서로 다른 수준의 모델링이 사용되는 예를 몇 가지 들어 보기로 하자.

- DUT(Design Under Test)는 아주 상세한 구현 수준(implementation-level)의 모델이지만, 테스트 벤치는 입력 값(Stimulus)의 생성과 출력값(Response)의 검사를 위한 추상적인 수준으로 작성되 사용되기도 한다.
- 처음부터 구현 수준의 상세 설계가 이루어졌더라도, 시뮬레이션의 속도 향상, 혹은, 세부 설계정보의 노출을 피하기 위하여 IP에 대한 상위 추상화 모델을 작성할 필요가 있을지 모른다.
- 시스템을 구성하는 주변의 모듈들은 상위 수준에 머무른 채, 특정 모듈에 한하여 상위 기능 수준의 사양에서 클럭-수준의 RTL 모델로 세부 설계될 수 있다.

특정의 모델을 현재 구현되어 존재하는 것 또는 가까운 장래에 실생활에서 구현될 것으로 여겨지는 것과 비교해 보면, 모델 표현의 구체화 정도에 대한 관점에서 서로 상이한 면이 몇 가지 있음을 알 수 있다. 이러한 모델링의 구체화 정도(modeling accuracy)를 여러 측면에서 살펴보자.

- 구조적인 구체성(Structural Accuracy) : 기술된 모델이 실제 구현하려는 시스템의 구조를 어느 정도 구체성을 가지고 기술하고 있는가를 반영한다. 예를 들면, 하드웨어와 소프트웨어로 분할되어 모델링 되었는지, 주요 하드웨어의 경우 실제 구현하려는 신호와 입출력 핀들이 해당 모델에 반영되어 기술되어 있는지, 소프트웨어의 경우 각 타스크간 통신이 실제 사용하려고 하는 실시간 운영체제(RTOS)에서 제공되는 통신 메커니즘의

수준으로 구체화되어 있는가 하는 정도를 의미한다.

- 타이밍의 구체성(Timing Accuracy) : 표현된 모델이 실제 구현하려는 시스템의 타이밍을 어느 정도 구체성을 가지고 기술하고 있는가를 반영한다. 타이밍 영향은 설계 사양을 정할 야기되는 요구 사항(Constraints)과 구현시 야기되는 처리 시간의 지연 때문에 발생될 수 있다. 타이밍은 절대적인 시간 단위(예를들면, seconds)으로 표현될 수 있으며, 종종 하드웨어를 다룰 때 클럭-싸이클 수준(연산에 사용할 수 있는 클럭의 갯수와 같은)으로 표현되기도 한다. 그 보다 더 낮은 수준(설계된 회로의 게이트 지연에 따른 동작속도 등)의 타이밍은 하드웨어 구현 툴(Synthesis Tools)의 도움을 받아 관리하기 때문에 크게 염려하지 않아도 된다.
- 기능적인 구체성(Functional Accuracy) : 표현된 모델이 실제 구현하려는 시스템의 기능을 어느 정도 구체성을 가지고 기술하고 있는가를 반영한다. 경우에 따라서는, 상위 수준 모델은 실제 구현에서는 있는 기능을 이지만, 모델을 단순화하기 위해서 또는 시뮬레이션의 속도를 향상시키기 위해서 모듈 내부의 복잡한 기능들을 생략하고 기술하는 경우도 있다. 예를 들면, 디지털 신호처리(DSP, Digital Signal Processing) 시스템의 설계시 개발 초기에는 Floating Point에 의한 모델을 작성하는 것이 일반적이다. 일단 알고리즘이 확실해지면, 하드웨어 구현을 위하여 연산의 정확도를 확보할 수 있는 수준에서 정수형, 혹은 비트 단위의 Fixed-Point 연산 모델을 작성한다.
- 자료 구성의 구체성(Data Organization Accuracy) : 표현된 모델이 실제 구현하려는 시스템에서 사용되는 자료 구성을 어느 정도 구체성을 가지고 기술하고 있는가를 반영한다. 예를 들어, 모델안에서 소프트웨어에서 사용하는 자료의 구조와 자료 영역 할당이 실제 구현될 시스템의 자료의 구조와 자료 영역 할당과 얼마나 잘 부합하는지 가늠해 볼 수 있다.
- 통신 프로토콜의 구체성(Communication Protocol Accuracy) : 표현된 모델이 실제 구현하려는 시스템에서 사용되는 통신 프로토콜을 어느 정도 구체성을 가지고 기술하고 있는가를 반영한다. 어떤 경우에는 상세 수준의 통신 프로토콜이 더 추상적이고, 효율적인 상위 수준의 프로토콜로 대체되는 것이 유용하다.

때때로 위에서 언급한 각 각의 모델링 구체화 정도의 양상들에 대해서, 모듈의 입출력 포트와 같은 모듈간의 경계에서 논하는 것인가 또는 계층적인 경우 상위 모듈 안의 모든 하위 모듈에도 이러한 구분을 확장할 것인가를 구분할 필요가 있다. 또한 위에서 언급한 모델링 양상들은 하드웨어뿐만 아니라 소프트웨어 모델에도 적용될 것임을 밝혀 둔다. 소프트웨어 모델 역시 구조, 타이밍, 기능, 자료 조직화, 그리고 통신 프로토콜의 측면에서 구분하는 것이 중요하다는 것을 알게될 것이다.

지금까지 모델의 구체성 정도를 판단할 수 있는 주요 양상들에 대한 구분이 이루어졌고, 지금부터는 다양한 모델링 수준를 기술하기 위해서 이 책에서 사용할 용어들에 대하여 살펴 보기로 한다.

Executable Specification은 어떤 제안된 구현 방법과는 아주 독립되어 설계의 의도된 기능(Design Specification)을 모델링하는 것이다. 만일 시간 지연이 실행형 사양(Executable Specification)에 표현되었다면, 이는 곧 구현시 지켜야 하는 시간의 요구 사항(timing constraints)을 나타낸 것과 같다.

Untimed functional model은 처리 시간의 지연에 대한 언급이 제외된 실행형 사양(Executable Specification)이라고 할 수 있다. Untimed functional model의 모듈 사이의 통신은 버스와 같은 공유된 통신 연결 모델이 없는 일대일 대응(Point-to-Point) 형태가 된다. 일반적으

로 통신(Communication)은 데이터 항목들이 모듈 사이에 신뢰성있게 전달되어질 수 있게 하기 위해 Blocking 읽기와 쓰기 방법을 가진 FIFO를 사용하는 것으로 모델링 된다.

Timed functional model은 모듈 사이의 통신이 버스와 같은 공유된 통신 연결 모델이 없는 일대일 대응(Point-to-Point) 형태이고 Blocking 읽기와 쓰기 방법을 가진 FIFO를 사용하는 것에서는 Untimed functional model과 같다. 다만, Timed functional model에는 사양(Specification)의 시간적인 요구 사항(Timing Constraints) 및 특정 타겟(Target)에 구현시에서 처리 시간의 지연을 반영하기 위해 설계안의 프로세스에 시간 지연이 추가된다. 게다가 시간 지연은 타겟(Target)에 구현의 통신 지연을 모델링하기 위해서 모듈 사이의 통신에 사용될 FIFO에 Annotate 되어질 수 있다. 시간 요소 기능 모델은 초기에 Hardware 및 Software Trade-Off 분석을 하기 위해 주로 사용된다. 이것은 프로세스를 하드웨어에 대입할 때 나타나는 시간 지연 형태와 프로세스를 소프트웨어에 대입할 때 나타나는 시간 지연 형태을 영향을 평가함으로써 이루어진다. 또한 Timed functional model은 Latency가 Control-Oriented Algorithms에 끼칠지 모를 영향을 평가하는데 사용될 수 있다.

사실 Executable Specification과 Untimed functional model Timed functional model은 실제 구현될 시스템의 구조와는 일치하지 않는 모델(Behavioral Model)일 수 있다.

Transaction level model에서 모듈간의 통신은 함수의 호출과 같은 방식을 취한다. 이러한 통신 모델은 전형적으로 기능의 정확성 혹은 시간의 정확성(때때로, 클럭 수준의 정확성을 나타낼 때도 있음)에 주안점을 두고 있지만, 구조적인 정확성을 띄지는 않는다. 예를 들면, SoC 플랫폼(System-on-Chip Platform)의 Transaction level model에서 온-칩 버스(on-chip bus)가 지원하는 버스트 읽고/쓰기 전송(Burst Read/Write Transactions) 등 여러 가지의 전송 형태를 모델링할 수 있지만, 버스에 연결되는 실제의 버스의 연결선 수(비트 폭), 버스에 연결될 모듈의 핀 따위들을 기술하지 않는다.

Platform Transaction-Level Model이라는 용어를 사용할 때에는 타켓 SoC 플랫폼(Target SOC Platform)의 통신 기반을 모델링하기 위해서 위에서 언급된 Transaction-Level 모델링 방법을 사용하는 것을 의미하며, 또한 타켓 SoC 플랫폼의 내부 블록과 구조적으로 상호 교신하는 통신 기반내의 모듈을 사용하는 것을 의미한다. Platform Transaction-Level Model들은 버스 로딩 및 쟁탈(bus loading and contention) 같은 것의 영향 및 전반적인 시스템의 성능을 정확히 모델링하기 위해서 사용될 수도 있다. 또한 Platform Transaction-Level Model들은 시스템 설계 초기에 하드웨어와 소프트웨어의 상호 작용을 모델링하기 위한 매우 빠르고, 정확하고 효율적인 방법을 제공한다.

Behavioral Hardware Model은 모델 간의 경계에서 입출력 핀(포트) 상세 수준의 모델이고, 모델 간의 경계에서 클럭-사이클 상세 수준이 아니라 기능적인 상세 수준의 모델이다. Behavioral Hardware Model은 실제 구현될 구조를 반영하는 내부 구조를 가지고 있지 않다. Behavioral Hardware Model은 하드웨어 행위 합성 툴(Hardware Behavioral Synthesis Tools)의 입력으로 사용될 수 있다 Pin-accurate, cycle-accurate hardware model은 모델 간의 경계에서 입출력 핀(포트) 상세 수준 및 사이클 상세 수준, 기능적인 상세 수준의 모델이다. 입출력의 측면에서 실제 시스템에 가까운 상세설계가 이루어진 것이다. 핀-싸이클 상세 모델(Pin-accurate, cycle-accurate hardware model)은 실제 구현될 구조를 반영하는 내부 구조를 반드시 가질 필요는 없다.

Register-transfer level model은 모델 간의 경계에서 입출력 핀(포트) 상세 수준 및 사이클 상세 수준의 모델이다. 게다가 Register-transfer level model의 내부 구조는 실제 구현의 레지스터와 조합 논리 회로를 정확하게 반영한다.

## Ⅳ. Unified Verification Methodology (UVM) 실현을 위한 Functional Virtual Prototype(FVP)

FVP(Functional Virtual Prototype)는 기술적인 기획자(Micro Architects)와 함께 일하는 검증팀에 의해서 개발 과정의 초기에 만들어진다. 일반적으로 검증팀은 System Level에서는 그들이 할 일이 별로 없기 때문에 RTL Code가 준비될 시점까지 프로젝트에 관여되지 않는다.

그러나 통합된 검증 방법론(UVM)에서는 검증 팀은 기술적인 기획자(Micro Architects)와 함께 일하면서, RTL Coding 전에 TLM (Transaction Level Model) 및 Transaction Level Testbench를 개발한다.

전체 설계 과정에서 FVP(Functional Virtual Prototype)가 바르게 사용되는지를 살펴보는 것 및 FVP(Functional Virtual Prototype)를 유지·보수하는 것에 검증팀이 가장 많은 이해 관계를 가지기 때문에, 검증팀이 FVP의 논리적인 주인이라 말할 수 있다.

다른 팀은 단지 FVP를 제한된 시간 동안 사용하고, 방치하곤 한다.

FVP는 TLM(Transaction Level Model)에서 출발하며 특정 용도의 요구에 맞게 상세하게 다듬어 지는데, 최종 구현물에 비교한 FVP의 충실도의 정도는 FVP의 사용에 목적이 의존한다.

예를 들면, 어떤 용도는 매우 상세한 모델을 요구하게 되고, 어떤 용도는 Software가 이용할 수 있는 Register 정도만 요구할 수 있다.

전형적인 설계 과정의 예로 FVP가 어떻게 사용되어지는가를 살펴보면,

1. TLM(Transaction-level) FVP가 기술적인 기획자(Micro Architects)와 함께 일하는 검증팀에 의해서 개발 과정의 초기에 만들어진다.
2. 만들어진 TLM FVP는 소프트웨어 개발, 구조적인 성능 평가에 즉시 사용될 수 있고, 설계 구현(Design-In) 팀에게 제공될 수도 있다.
3. FVP의 각 블록은 독립된 환경에서 개별 Core 팀에 의해서 세부적인 구현이 이루어진다. 이때 FVP의 TLM을 그들이 개발한 블록을 검증하기 위한 기준 모델로 사용한다.
4. 일단 블록이 독립된 환경에서 검증된 후, 그 블록은 FVP 안에 TLM 대신에 배치되어 시스템 환경에서 시뮬레이션 될 수 있다. 이를 통하여 시스템 요소의 통합에 관련된 잘못을 찾아 해결할 수 있다.
5. 일단 모든 블록이 각 각 FVP 안에서 검증된 후에는, 그들 모두가 합해져서 최종 구현 단계의 FVP(Implementation-level FVP)로써 검증될 수 있다.
6. 최종 구현 단계의 FVP(Implementation-level FVP)는 최종 소프트 웨어 개발과 설계 구현에 사용될 수 있고, 검증 중추로써 제공될 수 있다.

### 1. FVP(Functional Virtual Prototype)

그러면 FVP에 대해 좀더 자세하게 알아 보자. 문자 그대로, FVP는 서로 다른 Functional 도메인(Domains)을 포함하는 SoC 설계에 사용되는 기능적으로 동일한 가상의 System를 말한다.

FVP는 다음과 같은 요소로 이루어져 있다.

- Design modules : design module은 IC 안에 있는 단일 블록의 functional model을 의미한다. 이러한 functional model은 design 과정에 따라 다른 level의 abstraction으로 나타난다. Model은 개발을 하는 초기에는 transaction-level model에서 구현되며, 어떻게 사용 할 지에 따른 필요에 의해 세부적으로 기술되어진다. 결국, 이 model은 implementation-level model로 표현 되거나, 종종 RTL model로 coding 된다.
- Interface monitors : interface monitor는

design block들 사이에 정보의 이동에 대한 상태를 점검하여 올바로 Operation이 되는지를 검증한다.

- Testbench components : FVP에서 Testbench는 design 안으로 data를 입력하는 stimulus generator는 물론, design으로부터의 requests 응답에 대한 response generator, 그리고 FVP의 올바른 동작을 검증하기 위한 application checker를 포함한다. Testbench components는 design이 FVP에서 debug하는데 도움을 주며, 세부화 과정에서 사용되어지는 일반적인 test 환경을 제공한다.

Transaction-level FVP는 Implementation model보다 1000배 정도의 속도를 향상시켜 줄 수 있으며 Implementation보다 더 훨씬 일찍 소프트웨어의 개발을 가능하게 해준다. 이것은 소프트웨어 개발팀이 정확한 모델을 가지고 디자인 과정의 초기에 그들의 소프트웨어를 개발할 수 있도록 해준다.

FVP의 중요 개념은 디자인을 효과적으로 검증하기 위한 다양하고 혼합된 abstraction level을 사용, 디자인 팀간의 common model의 사용, methodology 안의 verification components의 재사용 등 세 가지로 정리된다. 또한, FVP를 만들 때 고려되어야 할 사항중에서 FVP에 대한 중요 세 가지 사용 목적은 소프트웨어 개발에 대한 executable model로써, Subsystem 개발과 integration에 대한 reference model로써, design-in team이라고 말하여지는 팀에 의해서 최종적으로 구현된 블록을 검증할 model로써 사용이 고려 되어져야 한다.

## 2. UVM을 적용한 4단계 설계 과정

UVM을 적용한 설계의 단계별 분류를 나누어 보면 다음과 같다.

### 1. SoC Design(SoC development team)

전형적으로 SoC Chip 개발 프로젝트는 FVP를 만들고 검증하는 SoC 개발팀과 함께 시작한다.

일단 FVP가 실제와의 충실도 측면서 만족할 만한 수준이면, 그것은 개별팀에게 제공된다.

### 2. Subsystem Design

Subsystem 개발 팀은 우선 그들의 Subsystem을 독립된 환경에서 검증한 후, FVP 안에서 검증한다.

각 Subsystem 개발 팀들이 그들의 Subsystem을 검증한 후, 그것을 SoC integration 팀에 제공한다.

### 3. SoC integration

SoC integration 팀은 검증된 Subsystem들을 함께 묶어, 부분 요소들의 통합과 관련된 것들을 시험한다.

### 4. System Verification

최종적으로, SoC Chip의 구현은 Testbench가 실제 환경(real-world environment)으로 대체된 상태로 system verification 단계에서 검증된다.

그러면 각 단계를 좀더 세부적으로 살펴 보기로 한다.

SoC Design 단계는 FVP의 창조에 집중된다.

설계는 이미 말했듯이 TLM(transaction level models)으로 시작되고, 새로운 디지털 모듈에 대해서는 TLM으로 시스템 모델링을 곧 바로 진행하고, 기존의 모듈이나 알고리즘 모듈에 대해서는 TLM이 핵심 기능부(Behavioral Core)을 transaction interface로 둘러싸는 형태로 만들어지며, Analog/RF 처럼 continuous domain 모듈에 대해서는 TLM이 존재하는 Behavioral models이나 C Level 알고리즘 형태로 만들어진다. Continuous domain 모듈은 Transaction-level interface에 연결하기 A2D(Analog to Digital) Conveter로 쌓여질 수 있다.

FVP를 디버깅하는 것은 매우 힘든 일이다. 확실히 FVP에 결함이 있다면, 설계에도 결함이 있을 것이다.

Interface monitors는 이용할 수 있는 Signal Level이 없을 때에 Transaction inter-

faces의 통신 규약을 감시하기 위해서 FVP 안에 위치한다.

Application checkers는 성능 감시, 기능적인 반응, 평가, 기타의 체계에 사용되는 specification, behavior, and processes를 검증한다.

Stimulus generation는 일반적인 testbench와 같다. Directed and Random type tests를 위한 인터페이스가 이용 가능하게 만들어지고, 둘 다 사용되어진다.

Assertions과 coverage와 같은 선진 verification technique들이 SoC design 단계에서 역시 사용되어질 수 있다. Application Level Assertion이 Application checkers에서 사용될 수 있고, Interface assertions이 Interface Monitors에서 사용될 수 있다. 이미 측정된 Application and Interface Coverage가 stimulus를 검증하기 위해서 사용된다.

일단 FVP가 완성되면, Subsystem 개발을 위해 subsystem team들에 전해진다.

각 subsystem team에 대해서는 후에 자세히 논하기로 하고, 여기서는 FVP가 독립된 Digital testbench 안에서 Top-Down 방식으로 어떻게 사용되어지는가를 살펴보기로 한다.

FVP의 Digital TLM은 그것과 Structural assertions를 포함한 Block-level implementation을 비교하기 위한 response checker에서 사용된다. Interface monitors 역시 약간의 수정으로 재 사용될 수 있다. 공통의 API로 쓰여졌다면 stimulus generators and transactors를 재 사용하는 것이 가능할 수 있다.

일단 Subsystem이 FVP와 격리된 상태로 검증된 후에, 검증된 Subsystem은 transactors의 사용을 통하여 Transaction level FVP에 Integration된다. 이때도 transactors와 Interface monitors는 재 사용될 수 있다. 모든 개별 Subsystem이 격리된 상태로 검증되고 Transaction level FVP에 결합되어 검증된 후, implementation-level FVP를 형성하기 위해서 함께 결합된다. 각 Subsystem 개발팀은 구현된 Subsystem 모듈을 어떤 필요한 Testbench 구성 부품(assertions, acceleration, interface monitors)과 함께 제공할 의무가 있다. 이 과정에서 중요한 단계는 transaction-level backplane(FVP)을 signal level backplane(FVP)으로 변경하는 것이다. 이와 같은 상호 연결의 검증이 매우 중요하기 때문에 가능한 빨리 transaction-level backplane(FVP)을 signal level backplane(FVP)으로 변경하는 것이 중요하다.

Subsystem이 *Digital* 블록, 알고리즘 블록, Analog/RF 블록, 소프트웨어 순서대로 또는 쉽고 효과가 많은 블록 순서대로 FVP에 결합되는 것을 제안하지만, 실제로 집적의 순서는 어떤 블록이 먼저 완성되느냐에 따른다. UVM의 최대 장점은 블록이 제공되는 순서에 상관없이 제공되는 즉시 FVP에 결합을 시작할 수 있다는 것이다.

System Verification은 종종 그 필요성이 경시되지만, Design 과정의 매우 중요한 부분이다.

System Verification은 Testbench에 녹아있는 모든 가설들이 타당한가를 검증하기 때문에 매우 중요하다.

디자인을 시뮬레이션하고 점검하기 위해 Testbench를 제거하고 실제의 Interface를 사용함으로써, 설계자는 사실상 Testbench를 검증하는 것이다.

UVM에서 System Verification은 세 가지 방법으로 실시한다.

1. 매우 느리지만 비교적 손쉬운 방법인 Simulation based system verification은 API를 통하여 외부 장비와 연결된 Simulation 모델을 계속적으로 사용한다.
2. 좀더 빠르고, 좋은 Debug 환경 그리고 가격 측면에서도 매력적인 Emulation based system verification은 Emulator를 사용하고, 그것을 표준 계측기 및 Debug 환경에 Interface한다.
3. 아주 빠르지만 FPGA에 익숙하지 않으면 아주 힘들고, 유지 보수하기도 힘든 Hardware

prototype system verification은 FPGA 보드를 사용하는 것이다.

## V. FVP를 구성하는 4-Subsystem Design

FVP를 구성하는 개별 Subsystem을 자세히 탐구해보고, 어떻게 UVM이 서로 다른 설계 영역들을 통합하는지를 살펴보자. 우선, Control-Digital을 살펴보자.

### 1. Control-Digital Subsystem Development

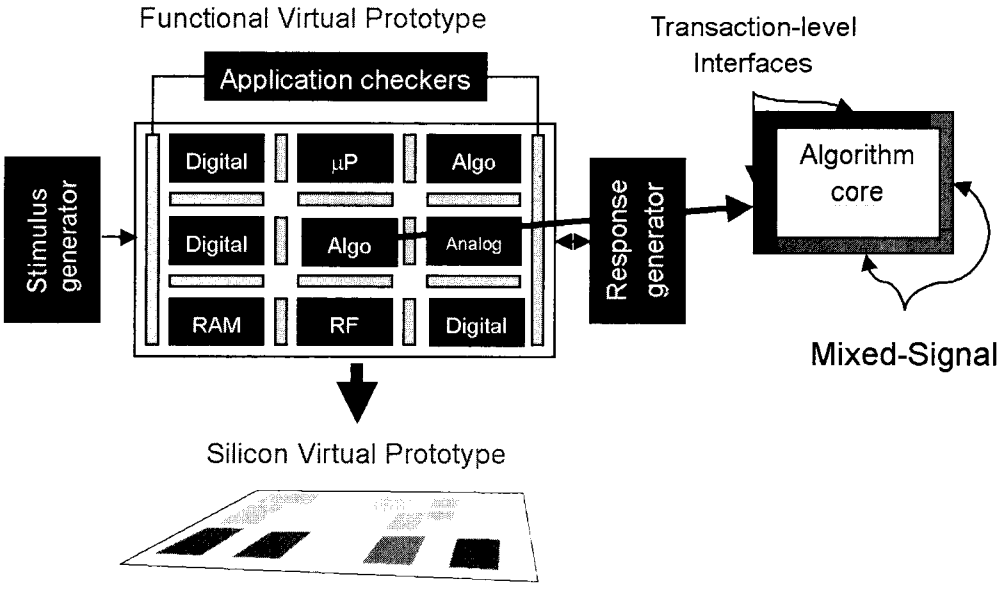
Control digital subsystems은 일반적으로 우리가 대규모의 디지털 영역이라고 고려하는 것이다.

Massive Digital ASIC에서와 마찬가지로 SoC의 control-digital subsystem의 독립된 검증을 다루는 것도 의미가 있다. UVM이 SoC에 중점을 두었기 때문에 디지털 설계자들에게 적절하지 않다고 생각할 수도 있다. 그러나 Control-digital 영역에만 관점을 집중해도, UVM은 많은 설계의 Issue을 해결해준다.

수년동안 큰 디지탈 설계의 전략은 디자인이 너무 커져서 전체로 검증하길 수 없게 되었기 때문에 디자인을 다루기 쉬운 부분으로 쪼게서, 개별적으로 검증하는 것이었다. 불해스럽게도, 설계 생산성 및 실리콘 용량은 급속도록 성장하여 검증이 따라 갈수가 없게 되었다. 대부분의 Respin은 기능 결함에 의한 것이라는 유명한 통계에서도 알 수 있다. 통계를 잘 살펴보고, 어디서 기능 격함에 빠지게 하는가를 살펴보면, 최대 2가지 이유로 귀결된다.

즉, 블록 사이의 Integration과 관련된 Issue와 specification의 잘못된 해석때문이다.

Integration 테스트는 앓는 엄지손가락처럼 도무지 어쩔 줄 모르게 하는 골치거리이다. 옛날 검증 방법은 각 블록을 위한 독립된 환경을 만드는 것이었다. 그런 후에 집적을 위한 분리된 환경을 다시 만드는 것이었다. 요즘의 유능한 검증 팀은 블록 수준의 Testbench의 일부를 집적화 수준에서 재 사용할 수 있게 Bottom Up 방식으로 그들의 검증 환경을 개발한다.




〈그림 2〉 Functional Virtual Prototype

Bottom-Up 방식의 예를 들면, 이것은 어려움 없이 바로 실현될 것처럼 보이지만, Bottom-Up 방식의 설계가 잘되기 위해서는 많은 계획과 의사 소통을 요구한다는 것을 경험에서 알 수 있다. 블록 수준의 Testbench을 위한 구성 요소을 개발하는 엔지니어는 그것이 Subsystem 수준에서 어떻게 사용되어질 수 있을지를 미리 생각해야 하고, 모든 필요한 특징들을 고려해야 한다. 물론, 그렇게 하는 것은 매우 어렵고, 많은 경우 나중에 블록들을 집적할때 부품을 수정하는 것이 필요하다.

UVM은 Bottom-Up 검증 방법이 오늘날 일부 팀들이 사용하기 위한 가장 좋은 방법일 수 있다는 것을 알고 있다. 그러나 SOC를 위해서는 Top-down 설계 및 검증 더 많은 장점을 가지고 있으므로 Top-down 접근 방법을 사용할 것을 제안한다.

Top-Down 접근 방법에서는 설계가 Subsystem에서 검증된 후, 블론 단위에서 검증되어진다.

우선 Subsystem 시험 환경을 만들어 그것을 검증한후에 블록 레벨 또는 하위의 검증 환경으로 세부적으로 구현되어 내려간다. 이와 같은 설계 방법에서는, 시스템의 구성 요소를 개발하는 사람은 전체 시스템을 항상 염두에 두기 때문에, 블록를 설계하고 검증하는 환경을 만들때도 아주 작은 세부화 과정만이 필요하게 된다.

Top-Down 설계 방법은 어떤 팀이 실제 구현을 여전히 Bottom-Up 방식으로 만들때는 퇴보하는 것처럼 보일 수 있다. 즉, 우선 블록을 만들고 검증한 후에 그것을 Subsystem에 집적하는데, 왜 처음부터 Subsytem 환경을 만들어야 하는가? 라는 반문을 할 수 있다.

그러나 Top-Down 설계 방법은 검증팀이 예상되는 문제를 미리 예측하여 준비할 것을 요구며, 블록이 RTL로 구현이어 사용가능하게 되기 전에 Subsystem이 작동되기를 요구한다. 그렇게 하는 것이 어떤 팀에게는 한 번은 해야 것이지만, 일단 그것이 작동하게 되면 그들은 검증을 위하여 반복되는 준비 과정을 거치지 않고 전체 설계 과정에서 그것을 계속 사용할 수 있다.

Testbench 개발의 Top-Down 방법은 FVP에서의 TLM들과 같은 기준 모델들에 의존한다.

## 2. Advanced Verification Techniques

Transaction Level의 Testbehch 개발과 더불어, 선진적인 검증 기술이 Control 중심의 Digital 설계의 전체 과정에서 사용되어진다.

UVM은 Assertion, Coverage, Transaction, Acceleration을 Digital Subsystem의 설계에서 뿐만 아니라 전체 System 설계 과정을 통하여 사용하기를 권장한다.

Assertion, Coverage, Transaction, Acceleration의 선진 검증 기법들이 Digital Subsystem에 검증에 집중되어 있고, 그 곳에서 가장 널리 사용되고 있지만, 다른 Embedded Software 및 Analog Digital Mixed Subsystem의 설계에서도 이와 같은 선진 검증 기술들이 아주 유용하게 사용되어진다.

Assertion, Coverage, Transaction, Acceleration의 선진 검증 기법들이 개별적으로는 단지 기법 또는 Tool이지만 함께 사용될 때는 통합 설계 방법론에서 중요한 부분을 차지한다.

간단하게 검증 기법을 살펴보면 아래와 같다.

Transaction Level Testbench는 엔지니어가 상위 추상 레벨에서 일할 수 있게 해주어 시뮬레이션, 디버깅 및 분석하는 것을 빠른고 간단하게 해준다.

Assertion 설계의 설계 의도를 파악하여 검증하는 것을 도와준다.

Assertion은 Architectural, Interface and Structural 크게 세가지로 분류된다.

앞서 말했듯이 Assertion은 설계자의 설계 의도가 정확히 반영되어 Coding되었는지를 파악하는 검증 기법이다. 모델링에서도 Gate 레벨에서 RTL 레벨로 올라가 Coding 하듯이, 검증도 HDL(Hardware Description Language) 레벨에서 PSL(property specification language) 레벨로 올라가서 검증하면 쉽고, 빠르게 그리고 효율적으로 검증할 수 있게 된다.

Interface assertion은 블록 사이의 Interface의 규칙을 검증하기 위해서 사용된다.

Architectural assertions(Application assertions)은 시스템의 구성 체계의 특성을 검증하기 위해서 사용되어 진다. 예를 들면, 전송을 위한 Bus Request 후에는, 반드시 Acknowledge 신호가 부여된다.

Structural assertions은 FIFO의 오버 플로우 또는 FSM의 부정확한 Transition과 같은 구현 단계에서의 하위 레벨 내부 구조를 검증하게 위해서 사용되어진다. 예를 들면, FIFO의 Read 및 Write 포인터가 서로 같은 위치를 가르키면, Data가 FIFO에서 읽혀질 때까지는 FSM의 State는 FIFOOverflow로 남아 있게 된다. Architectural assertions은 가능하다면 FVP로 부터 재사용된다. Interface assertions 블록의 경계 부분에서 FVP Interface Monitor로 부터 재사용된다. Structural assertions은 설계 팀에 의해서 코드 내부에 추가된다.

많은 경우 Coverage가 "이미 내가 검증을 위한 모든 것을 다 했나?"라는 질문에 답하는 방법이라고 잘못 생각되어진다. Coverage는 검증을 위한 작업이 언제 끝날까를 말해주지는 못한다. 다만, Coverage는 검증의 목표치에 도달하기 위한 가장 효율적인 방향으로 검증팀을 안내할 뿐이다.

Coverage도 각기 적당한 적용 영역이 있는 몇가지 Coverage 종류로 분류된다.

Application-level coverage은 Subsystem 집적 시험후에 측정되어 진다.

Interface-level coverage는 블록 그리고 Subsystem 테스트 후에 측정되어진다.

Structural-level coverage는 assertion은 사용하여 추가되어지고, 블록 레벨 테스트 후에 측정되어진다.

Code Coverage는 블록 테스트 후에 측정되어진다.

많은 사람들이 Functional Coverage를 생각하는데, 그것을 위한 방법으로 Transaction based coverage를 생각하게 된다. Transaction based coverage는 Interface coverage로 분류된다.

Acceleration은 관심 부분에 도달하기 위해서 오랜 시간이 걸리는 개별 시험을 아주 빠르게 하고, 반복되는 테스크를 아주 빠르게 수행한다. 통합 검증 방법론은 시뮬레이션 시간을 절약 방법들 중에서 Hardware Acceleration을 사용한 획기적인 Performance의 증대를 지원한다. 그러기 위해서는 Software based simulation environment에서 Hardware based accelerated environment으로 쉽고, 자연스럽게 넘어가는 방법이 제공되어야 한다. 통합 검증 방법론은 Hardware Accelerator를 사용하여 현저한 시뮬레이션 시간의 단축 효과를 볼 수 있을때 만 Hardware Acceleration를 사용하기를 권장한다. 오늘날의 설계 규모를 감안면, Hardware Acceleration을 사용하여 얻게되는 Performance 향상은 점점 더 매력적이 될 것이라 생각된다.

통합 검증 방법론은 ASIC 설계에서 단순히 낮 시간에 바뀐 부분이 synthesis 또는 timing에 문제를 야기하는지를 점검하기 위해서 매일 밤 logic synthesis를 돌리는 것과 유사한 방법으로 Hardware Acceleration을 사용하기를 권장하기도 한다.

## 3. Algorithmic Subsystem 설계 및 검증

Multi-media, DSP, Communications Application을 위한 Agorithmic based digital designs는 Top-down 설계 방법을 사용해 왔다. 즉, 무한의 자원을 사용하는 이상적인 환경에서 기능적인 측면만 고려한 Floating Point Algorithm 설계·검증 후에, Hardware 또는 Software의 사이즈를 고려한 fixed point algorithm로 세부 설계를 하여 검증한 후에 그것을 RTL 또는 Software를 구현하는 단계를 거치게 된다.

그러므로 algorithmic based digital design은 Top-Dow 설계 방법을 권장하는 UVM에 아주 잘 적용된다.

Algorithmic based digital subsystem에서 주요한 고려 사항은 정확하고 공통된 Testbech 환경을 구축해 하는 것이다. 많은 경우에 algorithmic based digital subsystem의 검증은 전송 환경(Transmission Channel)과 함께 많은 표준 시스템 구성 요소를 모델링하는 것을 표함한다. 대부분의 algorithmic based digital subsystem의 개발 과정에는 정확하고 방대한 규모의 표준 라이브러리를 요구한다.

Algorithmic based digital subsystem이 FVP에 집적될 때는 아래 그림과 같이 Digital Domain과의 Interface를 위해서는 Transaction-Level Interface로 analog domain과의 Interface 부분에는 mixed-signal interface로 처리한다.

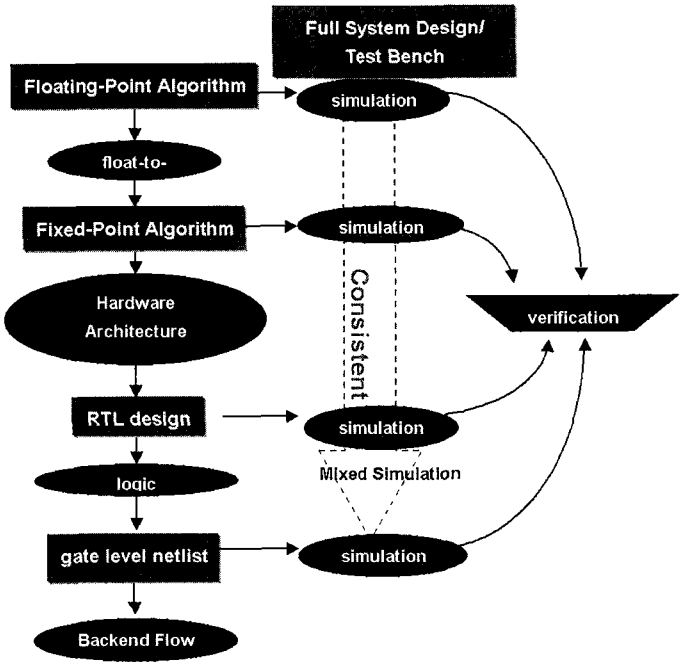
Testbench 환경은 Floating, Fixed, RTL, Gate 등 여러 abstraction Level에서 Algorithm을 검증하는 것을 지원해야 한다. 이것은 기능적인 정확성의 검증뿐만 아니라, 세부화 과정에서의 정확성 손실이 여전히 수용 가능한 정도인가를 검증하는 것도 요구한다. 이런한 검증은 bit error rate monitors, constellation diagrams 등과 같은 많은 다양한 가상의 계측기를 사용하여 수행된다.

Algorithm이 Subsystem에서 만족할 수준으로 검증된 후, 그것을 FVP에 집적하는 작업이 시작된다.

Algorithmic susbystem은 일반적으로 기능성 검증을 위해서 behavioral models을 사용하는 analog/RF Subsystem에 집적되어, Mixed Mode 시뮬레이션이 수행된다. Algorithmic subsystem은 간단한 Register file에 연결을 통해서나 PCI와 같은 Bus Interface를 통하여 control-based environment에 집적된다.

이것은 FVP에서 TLM을 사용하여 수행된다. 일단 Algorithm이 analog/RF Subsystem 및 control-based Subsystem과 집적된 후는 FVP 안에서 검증이 이루어질 수 있다.

시스템 엔지니어들의 작업은 주로 목표로 하는 제품의 standard에서 출발한다. 이러한 문서들




〈그림 3〉 Algorithmic Subsystem Design Flow

은 주로 IEEE, ITU, ETSI 또는 관련 기관들을 통해 얻게 된다. 문서에 제시된 요건을 충족하며 실행 가능한 스펙의 명세서를 만드는 것은 중요한 작업이다. 시스템 설계에는 많은 경우 floating-point의 high level model 라이브러리가 사용된다. 이러한 model들은 Standard에서 기술한 기능과 성능 요건을 충족시켰음을 증명할 수 있도록 하기에 중요하다. 디자인이 만족스러운 시뮬레이션 performance를 보이고 나면, 엔지니어는 설계에서 architectural constraints를 찾기 시작한다. Datapath에서 정확한 수의 비트를 찾는 것은 하드웨어 개발에 있어 타이밍, 전력소비와 area의 조절에 있어 매우 중요하다. 설계는 이제 fixed point로 변형되고, 필수적인 분석이 끝나고, 설계된 디자인이 성능 기준 요건에 만족 함을 증명하게 된다. 분석이 끝나고 나면, 설계의 하드웨어 구조에 대한 결정이 내려져야 한다. 대개의 경우, 이러한 결정은 시스템 엔지니어와 ASIC 엔지니어의 공동 작업에 의해 내려진다. 위의 그림처럼 시스템 시뮬레이션의 Flow가 구성되었고 엔지니어들은 하드웨어 디자인을 검증하기 위해 테스트벤치로 사용할 벡터를 만들고 시뮬레이션을 수행한다. 이러한 접근법은 테스트벤치가 non-reactive고, 채널(channel)과 같은 boundary condition이 고정적일 때는 성공적이다. 위의 그림에서 보여지는 것과 같이 flow의 모든 단계가 확장된 테스트벤치로 실행될 것이다(다시 말해 Coverage와 인터페이스 분석을 함께 제공한다). RTL 설계 단계 뒤로, synthesis, floor planning, place and route, 그리고 물리적 레이아웃이 뒤따른다.

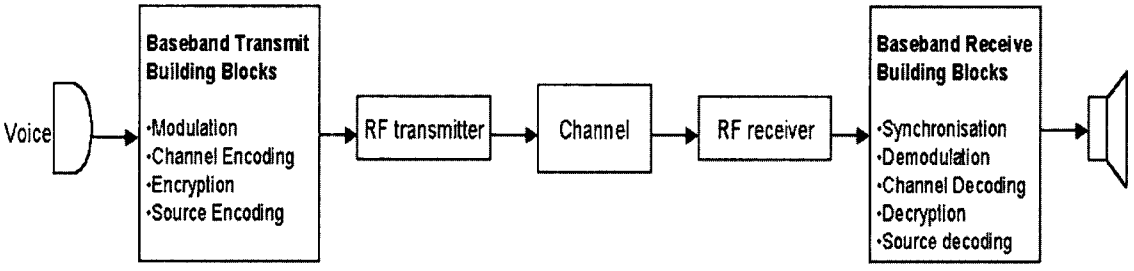
## 4. Analog/RF Subsystem의 설계 및 검증

SOC 플랫폼 기반의 chip 개발에서 빼놓을 수 없는 또 하나의 포인트는 디지털과 아날로그를 포함한 전체의 검증이다. 그를 위한 FVP는 다음 세 가지 통합 통합 시뮬레이션 즉, 첫 번째, Behaviora Level, Transaction-Level과 Implementation level 간의 통합, 두 번째 하드웨어와 소프트웨어 간의 통합, 마지막으로 아날로그와 디지털간의 통합 시뮬레이션이 가능하여야 한다.

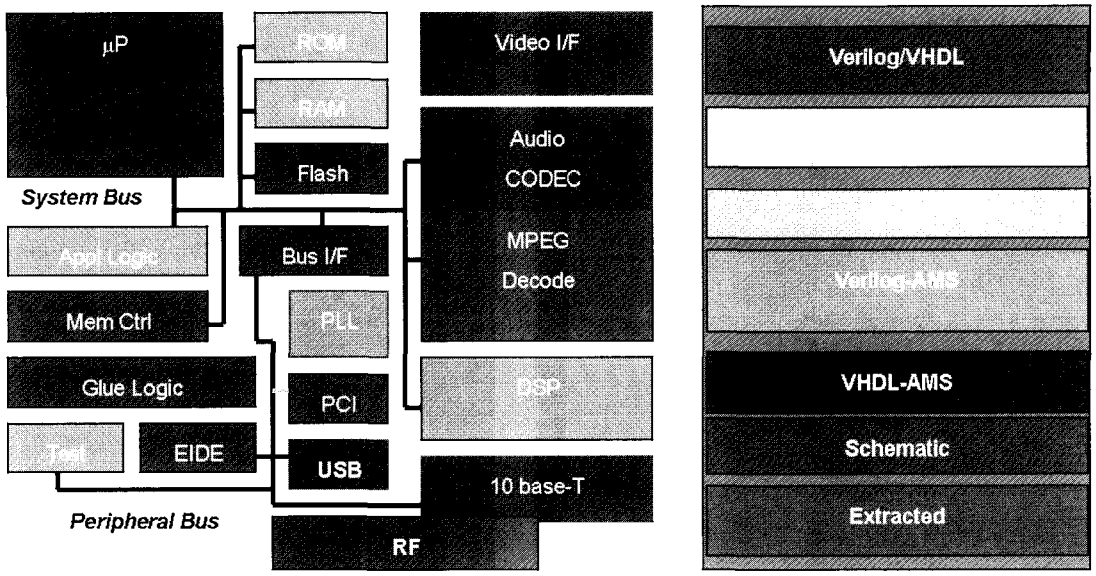
오늘날의 Analog 검증은 여전히 Transient와 AC effects에 초점이 맞추어져 있다. 그러나 Digital과 Analog 사이의 Integration은 통합 검증을 요구할 만큼 충분히 복잡하게 되었다. 통합 검증은 black-box를 사용해서는 불가능하게 되었다. Mixed Simulation을 위한 High level(C based) algorithmic Analog 블럭은 너무 High level이고, Transisitor 또는 netlist은 너무 low-level이다.

따라서, Digital을 충분히 빠르게 시뮬레이션하고, Analog를 충분히 정확하게 시뮬레이션하기 위해서는 Behavioral model이 반드시 필요하게 된다. Control based digital과 유사하게 오늘날 Analog design도 Top-down 및 Bottom-up 접근에 대한 논의가 있다.

일반적으로 Analog 설계는 이미 존재하는 설계의 부분적인 변경이거나, 단순히 새로운 Fab Technology에 적용 또는 다른 파라미터의 사용을 위한 변경이다. Analog 설계에서도 작고 잘 알려진 시스템 설계에는 Bottom-up이 잘 어울리지만, 시스템 설계는 behavior model로 시작




〈그림 4〉 Analog and Digital Mixed Design




〈그림 5〉 Example of SOC with mixed Level and mixed signal

되는 시스템 분석이 요구된다. Top-down 방식은 Behavior model로 출발하여, 시스템의 구성 요소로 세부적인 설계를 진행한다. 어떤 Analog 설계 방법이건, Digital 및 Analog Domain 사이의 상호 접속 및 기본적인 작동을 검증할 필요가 있다.

위의 그림에서 보면 하나의 칩 안에 다양한 서로 다른 언어로 구현된 디지털과 아날로그 블록이 혼재되어 있다. IC 공정기술이 발전하면서 전체 시스템은 수백만, 수천만 게이트 이상으로 커졌으며 초고속의 동작 주파수에서 작동되기도 한다. 거의 모든 경우 디지털 개발팀과 아날로그 개발팀은 별도로 존재한다. 이러한 환경에서 과거의 방식, 즉 상호간의 악영향이나 인터페이스 문제가 완전히 검증되지 않은 상태로 디지털부와 아날로그부가 완전히 서로 독립적으로 개발되고 검증되어 Mixed SOC chip를 개발한다면 처음의 시도가 성공할 가능성은 현저히 떨어진다. 문제는 디자인 Re-spin에 들어가는 비용이 이제는 너무 커져 버렸기에 단번에 성공할 가능성을 높이는 아날로그 디지털 통합하는 FVP에 관심을 둔 새로운 설계 방법론이 필요하게 되었다. 아날로그와 디지털 회로 간의 인터페이스 문제 등을 검증하기 위하여서는 첫 번째로 디자이너들은 자유롭게 디지털과 아날로그, Behavioral과 schematic, RC extracted view나 digital timing view를 같이 시뮬레이션하고 검증할 수 있는 환경이 필요하다. 두 번째는 시스템 function의 대부분을 차지하는 디지털부의 설계시 아날로그의 Non-linear 특성을 고려할 수 있도록 behavioral analog modeling 작업이 필요하다. TR level의 시뮬레이션은 대단히 정확하지만 모든 회로가 완성되고 공정 파라메터가 정해진 경우에만 시뮬레이션이 가능하고, 무엇보다도 RTL에 비하여 시뮬레이션에 대단히 많은 시간이 걸리기 때문이다. Verilog-AMS나 VHDL-AMS로 모델링 된 아날로그는 충분한 Function 검증이 가능한 수준으로 디지털(SystemC, C/C++, Verilog, VHDL)과 통합 시뮬레이션에서 만족할 만큼의 스피드를 얻을 수 있고 디자인 Refinement 과정을 거치면서 Spectre/Spice netlist와 같은 실제의 세부 모델로 구현되어 간다.

## Ⅵ. UVM으로의 점진적으로 이행

통합 검증 환경으로의 설계 환경을 변경하기 위해서는 많은 것들이 고려되어야 하는데, 보유하고 있는 자원(인적, 물적), 숙련도, 검증에 대한 필요성 및 그 정도등을 고려하여 이행 속도의 정도를 정해야 한다. 또한 이미 보유하고 있는 설계 환경이 가장 큰 고려 사항이 될 것이다. 몇 가지 시나리오를 생각해 볼 수 있다.

Control-Digital Subsystem, Algorithmic Subsystem, Analog/RF Subsystem 등의 개발 환경을 위한 Design Flow를 System Spec. 에서 실제 구현까지 통합된 환경에서 설계할 수 있으며, 또한 2개 이상의 Domain(예를 들면, Control-Digital Subsystem와 Algorithmic Subsystem 또는 Algorithmic Subsystem과 Analog/RF Subsystem)을 통합된 설계 환경에서 개발 할 수 있을 것이다.

그럼 아래와 같이 UVM으로 점진적인 이행 계획을 세워보자.

1. 서로 다른 프로젝트, 서로 다른 Domain, 서로 다른 검증 과정, 서로 다른 Design Chain의 엔지지어로 Verification methodology team을 만들고
2. UVM에서 차별화된 가치를 확인하여
3. 과정에서 고유한 가치를 제공하지 않는 것은 표준화하고…
4. UVM을 지원하는 Platform으로 설계 환경을 변경하고
5. 가장 실천 가능한 것부터 UVM으로 옮겨간다.

## Ⅶ. 끝 맺으면서

일반적으로 SoC 설계는 독립된 팀들에 의해 이루어지고, 각 팀은 그들만의 검증 기법을 사용해 왔다. SoC가 서로 독립되어 있어 협력관계인 IP 팀, 내부 핵심 팀, 알고리즘 개발 팀, Analog/RF팀 그리고 전단계 팀에서 얻어진 결과들로 이루어질 경우, 시스템 요소의 통합은 쉽지 않다. 또한, SoC 검증팀이 시스템 요소의 통합을 테스트하기 위해서는 전혀 새로운 검증 환경을 만들어야 한다. 이런 문제점들이 검증 Speed와 Efficiency를 떨어지게 하며, 이는 설계환경이 조각 조각 나누어져 있는데 기인한다. 검증 과정에서 각 단계별로 시스템 레벨과 아키텍쳐 레벨 등으로 나누어지고 서로 다른 도메인(디지털과 아날로그, 하드웨어와 소프트웨어) 프로젝트 사이의 구분은, 설계시 사용하는 호환성 없는 언어와 툴, 기술에서 시작된다. 이런 문제를 해결하기 위한 유일한 길은 통합된 검증방법론으로 전체 검증 과정을 단일화하는 것이다. 통합검증이란 하나의 디자인, 시뮬레이션 환경에서 한 부분도 빠짐없는 전체 시스템이 시뮬레이션 스피드의 부담없이 돌아가는 것으로, 많은 시간을 검증이 아닌 설계과정에 투자할 수 있으며 보다 완성도 높은 디자인을 더 빠른 시간에 시장에 내어놓을 수 있는것이다. 본고에서는 통합 검증방법론의 전체적인 흐름 및 툴의 구조에 대하여 기술하였다.

### 참 고 문 헌


〔1〕 SystemC를 이용한 시스템 설계(System Design with SystemC) (에이콘, 역 ; 국일호, 저 ; Thorsten Groter, Stan Liao, Grant Martin, Stuart Swan)

〔2〕 Surviving the SOC Revolution (A Guide to Platform Based Design) (Kluwer, 저 ; Henry Chang, Larry Cooke, Merrill Hurt, Grant Martin, Andrew McNelly, Lee Todd)

〔3〕 *http://www.cadence.com/products/incisive.html*의 Application Note 및 White Paper

## 저 자 소 개

**김 규 흥**

1994년 2월 중앙대학교 전자공학과 졸업, 1993년 11월~1997년 2월 : (주)한화/전자 정보통신, 1997년 2월~현재 : 케이던스 코리아, 〈주관심 분야 : Platform Based Design, 통신 Algorithm 개발 및 구현〉

연락처 : (회사) 031-728-3070

E-mail : khkim@cadence.com