



Hacia una metodología de diseño avanzado

Estrategias y propuestas para
optimizar un flujo de diseño de ASICs

Alfonso Chacón Rodríguez



Bibliografía

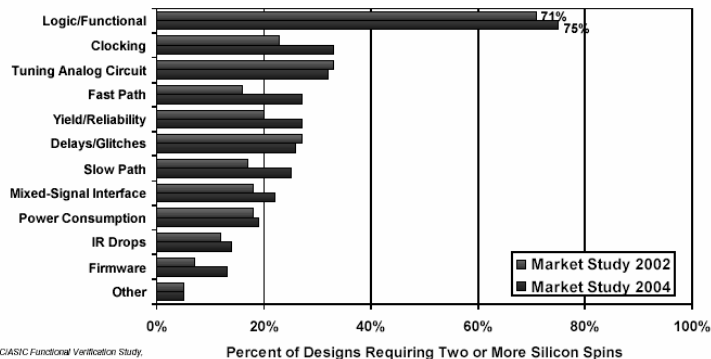
- Keating Michael and Bricaud Pierre,
*Reuse Methodology Manual: For
Systems-on-a-Chip Designs.*
Massachusetts: Kluwer Academic. 2002

¿Por qué pensar en una metodología de verificación?

- Diseños de millones de transistores
- Miles de casos de prueba a verificar
- 50% de los chips diseñados hoy en día requieren dos o más fundidas
 - 74% de estos re-spins se deben a errores funcionales
 - 60-80% del esfuerzo de desarrollo se emplea en verificar

¿Por qué pensar en una metodología de verificación?

IC/ASIC Designs Requiring Re-Spins by Type of Flaw

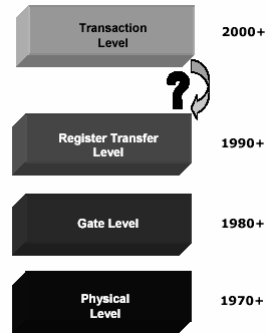


¿Por qué pensar en una metodología de verificación?

- Niveles de abstracción cada vez mayores

`always@(decade change)`
`abstraction.raise();`

- Estamos cambiando de un enfoque RTL a un enfoque TLM
- La verificación está en el camino crítico. Hay que tomarla en cuenta.



¿Qué entendemos por verificación?

- La verificación vs. el testing
 - El testing se hace para asegurar que un diseño fue fabricado correctamente
- La verificación no es escribir TBs. La verificación debe determinar
 - Qué patrones de entrada suministrar al diseño
 - Cuál es la respuesta esperada

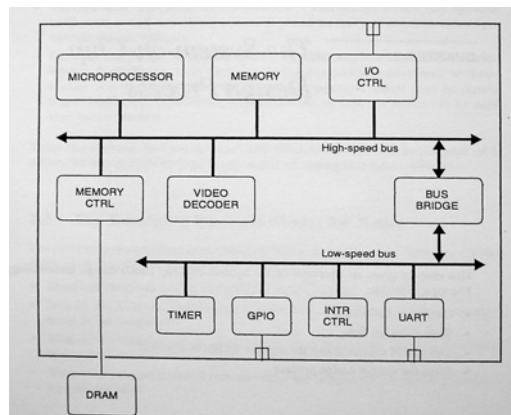
¿Qué entendemos por verificación?

- La verificación vs. validación
 - La verificación se hace para asegurar que la implementación cumple con la especificación. La validación verifica la especificación.
- La verificación vs. WCCA.
 - Un análisis de peor caso es una determinación cuantitativa del rendimiento del equipo, considerando su fabricación, el ambiente y los efectos de envejecimiento.
 - Se busca verificar la robustez bajo condiciones extremas

La estrategia típica

Tenemos un complejo sistema SOC

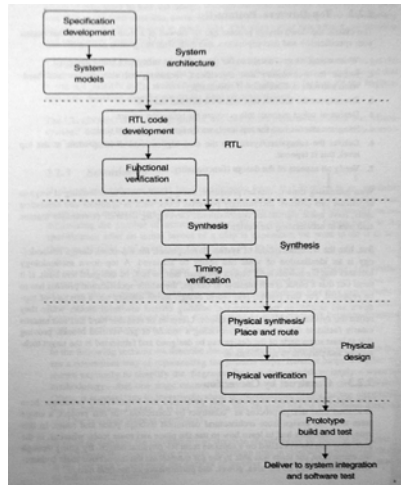
Algunas partes son Soft Macros, otras Hard Macros.





La estrategia típica

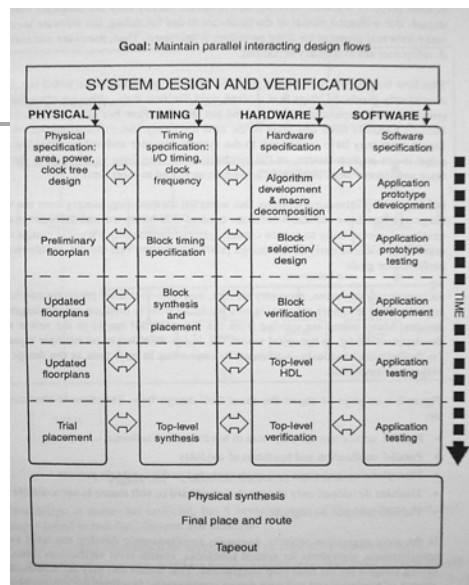
En el diseño tradicional, seguimos una especie de cascada



Otra estrategia

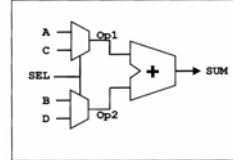
Lo ideal, es tratar de ir trabajando en paralelo.

Para ello, las especificaciones de diseño y de verificación deben estar claras desde el arranque



Definiciones

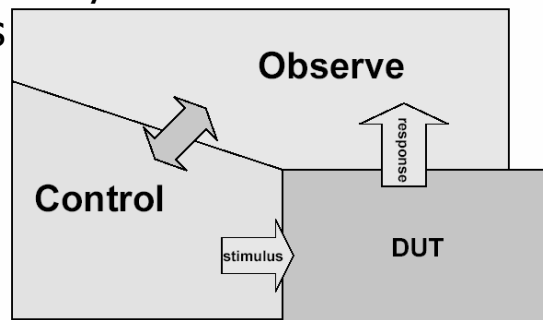
```
if (SEL == 1'b1)
begin
  Op1 = A;
  Op2 = B;
end
else
begin
  Op1 = C;
  Op2 = D;
end
SUM = Op1 + Op2;
```



- El objetivo de la verificación es asegurar que un macro (soft o hard) sea 100% correcto en funcionalidad y temporizado.
- Los soft-macros son bloques IP empaquetados para ser reutilizados, incluyendo su ambiente de verificación. Pueden venir en formato esquemático o, preferiblemente, en un HDL/HVL.
- Los hard-macros son bloques IP que se entregan en formato GDSII. Sus posibilidades de reutilización son limitadas.

El problema de la verificación

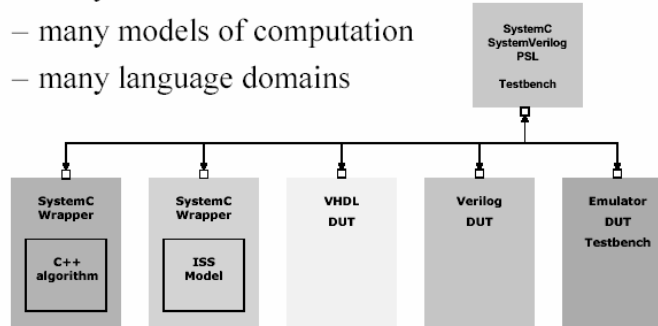
- Debemos ser capaces de estimular la unidad bajo verificación y observar sus respuestas.





El problema de la verificación

- Typically, in one simulation, we integrate
 - many abstraction levels
 - many models of computation
 - many language domains



Hay que tener un método

Key Aspects of a Good Methodology

- Automation
 - Let the tools do the work
- Observability
 - Self-checking is critical
 - Automate self-checking and coverage tracking
 - Assertion-based Verification
- Controllability
 - Make sure you exercise critical functionality
 - Constrained-Random stimulus and formal verification automate the control process
- Verification Planning and coordination
- Reusability
 - Don't reinvent the wheel
 - Reusability is functionality- and/or protocol-specific
 - Reuse is critical across projects, across teams, and across tools
- Measurability
 - If you can't measure it, you can't improve it
 - Tools automate the collection of information
 - Analysis requires tools to provide information in useful ways
 - All tools must consistently contribute to measurement and analysis



Un plan de verificación

- Arrancar con una especificación (un golden reference)
 - El plan debe definir
 - Estrategia de testing
 - Definición del ambiente de verificación
 - Herramientas de verificación a usar
 - Lista de los casos de prueba específicos
 - Definir límites de pruebas (hasta cuándo estaremos satisfechos)
- El ambiente de verificación es el conjunto de componentes TB a usar:
 - Modelos funcionales (tanto RTL como de comportamiento, los últimos generalmente provistos del golden reference)
 - Monitores de bus
 - Modelos de memoria
 - Conexión estructural con la DUT



Estrategias de verificación

- Verificación de sub-bloques (que las unidades mínimas sean funcionales: compuertas, registros, muxes, etc.)
- Verificación de macros (que los macros sean funcionales: UARTs, sumadores, etc.)
- Definición de tests de verificación (a ejecutar en simulación, con otras herramientas y sobre un prototipo)
 - Compliance testing. Contratar entre respuestas del RTL y la golden reference con los tests corridos para la validación de la última
 - Corner cases. Definir los casos extremos (en términos de velocidad, consumo, carga de la arquitectura, etc.)
 - Tests aleatorios
 - Tests con código real (para SOC)
 - Tests regresivos (una vez que se corrige un error, el test que lo encuentra se une a la lista de tests efectuados por defecto, y estos se vuelven a ejecutar de nuevo)
 - Chequeo de propiedades (más adelante)



Estrategias de verificación

- Herramientas de verificación:
 - Inspección
 - Simulación (usando TBs)
 - Usar lenguajes de verificación (TBs más complejos escritos en HVL)
 - Proveer de herramientas de control de versiones (CVS) para mantener la coordinación del diseño
 - Usar hasta donde sea posible herramientas de linting, code coverage y property checking
 - Prototipado (en FPGA)



Inspección de código y esquemáticos

- Lo más rápido y barato para encontrar errores
- El diseño es presentado por una parte del equipo. El resto del equipo revisa a mano el código o los esquemáticos.
- Antes de pasar a la primera simulación en lote, se puede hacer una simulación a paso sencillo, desde un estado conocido de la DUT, con break points escogidos, y estudiar la evolución del sistema



El testing adversativo

- La parte del equipo que hizo el diseño hace la verificación básica.
- El resto del equipo plantea una estrategia de verificación que trate de romper el diseño (siempre, por supuesto, desde las especificaciones).



Simulación y TBs

- Principios básicos para un buen TB
 - Los estímulos deben generarse, y las respuestas revisadas, en términos de transacciones.
 - El TB debe usar partes reutilizables
 - Todo el chequeo de respuestas debe ser automático (no revisando formas de onda)
 - Una transacción es una secuencia de acciones en un pin o bus



Simulación y TBs

- Chequeo de salidas

- Son las transacciones legales?

- Usamos monitores

```
//synopsys translate off
```

```
always@(posedge dma_clk)
```

```
if (fifo_full && fifo_write)
```

```
$display("DMA buffer overflow at time  
%d", $time);
```

```
//synopsys translate on
```

- Son las transacciones correctas?

- Comparamos contra el golden model



Simulación y TBs

- Verificación basada en componentes

- Se provee de una capa esencial de abstracción

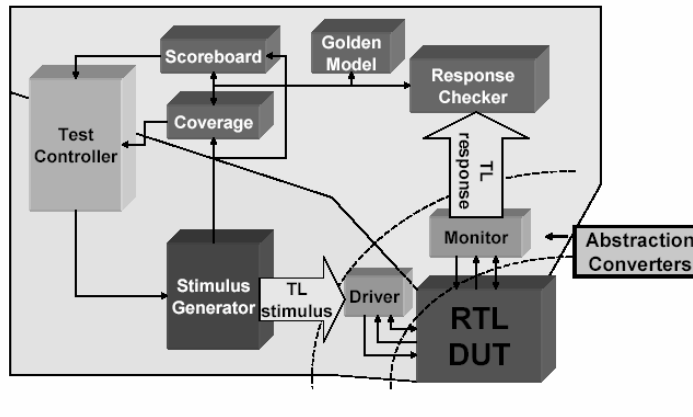
- Se pueden usar comandos de alto nivel sin preocuparse por los detalles de la DUT

- Usamos monitores de protocolo y de cobertura

- Muy importante: necesitamos saber qué se ha probado y qué no.

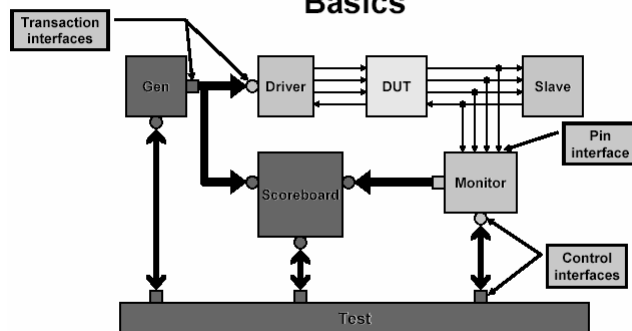


Simulación y TBs



Simulación y TBs

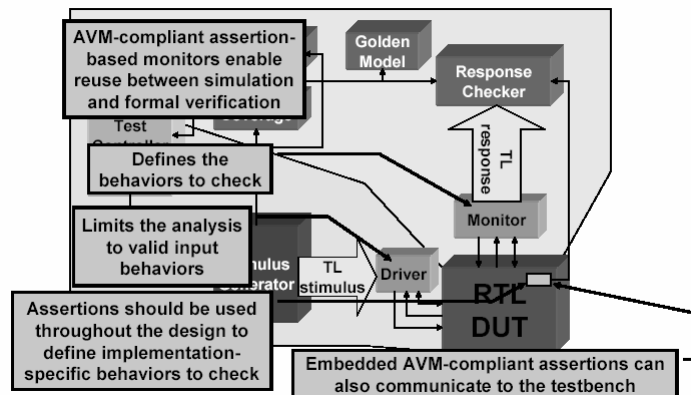
Building a Testbench Basics



Intent is captured in test and scoreboard



Diseño de un suite de TB



Diseño de un suite de TB

- Una vez listo el TB, hay que diseñar los tests suites
 - Compliance test: Usando el golden reference como base, hay que diseñar un test para CADA UNA de las funciones de la DUT. Esta es la base de la funcionalidad correcta del macro
 - Corner Cases: Condiciones de límite



Diseño de un suite de TB

- Test aleatorio: Con los dos métodos anteriores, se dejan afuera muchos posibles escenarios. La mejor manera de alcanzar la máxima confianza en el diseño es complementar esto con un test aleatorio.
- Cobertura funcional. Es un refinamiento del proceso anterior, muy usado por los HVL como System Verilog y SystemC. Se lleva una contabilidad de la cobertura de las regiones del diseño (cuánto han sido probados) y prosigue el análisis aleatorio sobre las zonas que tienen poca cobertura.



Diseño de componentes de verificación

- Lleva mucho tiempo al inicio, pero es la clave para una verificación robusta reutilizable.
- Es muy usado en la industria y existen ya muchas prácticas documentadas para diseño de modelos funcionales y monitores.



Diseño de componentes de verificación

- Modelos funcionales de bus
 - Servicios estándar
 - Control de secuencias
 - Servicios de mensaje y de bitácora
 - Con más capas de abstracción
 - Señal
 - Comando
 - Generación de tráfico



Diseño de componentes de verificación

- Modelos de dispositivos. Interactúan con la DUT.
 - Ejemplo: se está diseñando un controlador SDRAM. Se necesitan modelos funcionales de SDRAM.
- Monitores
 - Chequeo de protocolos
 - Control de secuencias
 - Servicios de mensaje y de bitácora
 - Reporte de cobertura



Verificar el temporizado

- Verificación de temporizado
 - Vital para asegurar cumplimiento de las especificaciones
 - Simulaciones a nivel de compuertas con información postsíntesis (o post layout) de las bibliotecas de la tecnología ofrecen una aproximación inicial.
 - El estándar en la industria es la verificación estática de temporizado STA(static timing analysis).
 - En STA se busca el worst-case delay del circuito para todas las posibles entradas. Con esta información, se verifica el cumplimiento de los tiempos críticos del diseño (hold, setup time de registros, máximo período de reloj esperado, etc.). PathMill de Synopsys es la herramienta más usada.



Cómo llegar al 100%

- Cobertura de código
 - Aunque esto no asegura que la funcionalidad de un código ha sido verificada, alerta sobre secciones del código sin revisar. Es por tanto una herramienta útil para diseños muy grandes.
- Prototipado en FPGA
 - Mucho más rápido que simular, permite además una verificación a nivel sistema.
 - Si el chip forma parte de un sistema, y si debe correr un software X que está siendo desarrollado en paralelo, esta es la forma más eficiente de asegurar que tanto la gente de software como la gente de hardware están cumpliendo con la especificación



Cómo llegar al 100%

- Chequeo de propiedades (o uso de aserciones).
 - Puede que algo quede sin cubrir
 - Usando métodos formales, puede cubrirse aquello que los casos de testing dejan al descubierto.
 - Usando lenguajes como SystemVerilog, o PSL para VHDL/Verilog, pueden incluirse chequeos formales para asegurar, por ejemplo, que un FIFO no sea nunca leído cuando está vacío.



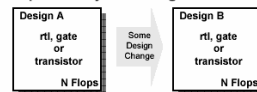
Cómo llegar al 100%

- La verificación de casos es un chequeo de equivalencia (contra la golden reference). El chequeo de propiedades verifica que el comportamiento del chip sea adecuado.

Equivalence vs. Property Checking

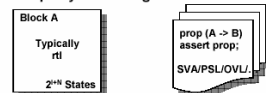
- Design A = Design B?
 - Chip level runs
 - Specific Algorithm
 - Explores equivalency of cones of logic between states
- Assertion = Design Behavior?
 - Block level runs
 - Many Algorithms
 - Exhaustive state space exploration of the block

Equivalency Checking



Pass: Design A = Design B
Fail: Design A != Design B
Debug shows cones of logic that aren't equal

Property Checking

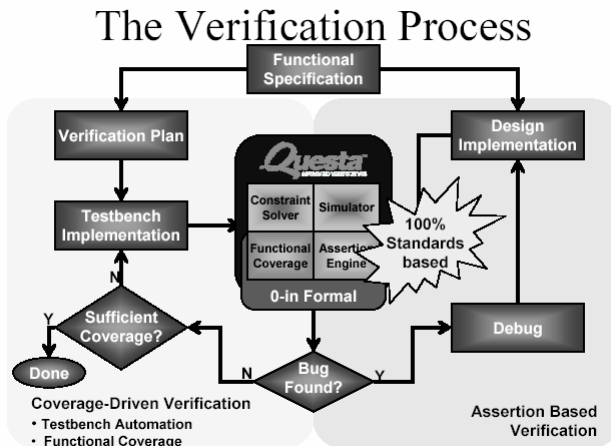


Proof: Formal Proof for all input conditions
Firing: Some condition violates property
Debug gives counter example showing firing



Cómo llegar al 100%

- En resumen



Ahora sí: ¿cómo hacemos esto?

- Establecer un flujo de diseño que incluya la verificación como parte integral
- Iniciar el plan de la verificación en paralelo con el diseño
- Crear siempre un golden reference, preferiblemente en un lenguaje simulable (C o un HDL/HDL)
- La especificación (golden reference) debe estar completamente validada al arrancar el diseño del chip

Ahora sí: ¿cómo hacemos esto?

- Crear un cronograma rígido para los diseños. No se pasará a otra etapa hasta que el equipo esté de acuerdo en que las etapas anteriores han sido satisfactoriamente cumplidas.
- Diseñar los macros usando guías establecidas para que sean reutilizables.
 - Estandarizar nombres de señales
 - Usar modelos, variables parametrizables
- Crear una biblioteca reutilizable de partes, TBs, código, completamente documentadas.
- Efectuar pruebas de sistema a nivel de prototipo para asegurar funcionalidad.

Ahora sí: ¿cómo hacemos esto?

