Tema 7: Métodos de Verificación Formal para Circuitos Digitales

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Acknowledgement to Matt Venn (YosysHQ) & Dan Gisselquist

Contexto docente

BT03: Verificación funcional y formal de circuitos digitales

- Tema 6: Capacidades de verificación funcional en circuitos digitales
- Tema 7: Métodos de verificación formal para circuitos digitales
- Tema 8: Diseño de planes de pruebas

Conocimientos previos requeridos:

- Conocimientos básicos de VHDL
- Assertions

Objetivos de aprendizaje

- Conocer las limitaciones de la verificación basada en testbenches
- Conocer las capacidades y limitaciones de la verificación formal
- Saber describir las suposiciones de un circuito al respecto de sus entradas
- Saber describir las propiedades de un circuito al respecto de sus salidas y estados internos
- Comprender los métodos Bounded Model Checking y k-induction

Repaso

Capacidades de Verificación Funcional:

- Conceptos para transformar un test dirigido en un test estructurado que se auto-chequea
- Métricas de verificación
- Modelado a nivel de transacción
- Assertions

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Donde los testbenches no llegan

" A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 9999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone. "

Donde los testbenches no llegan

- Test ‘constrained random’ incrementa la confianza en el diseño, pero…
  - Puede demostrar que no hay bugs?
- Estamos seguros de que hemos probado todos los casos posibles…
  - o sólo los que se nos han ocurrido?
- Normalmente al simular probamos los casos en los que todo va bien (funcionalidades esperadas)
  - Pero no solemos ser tan creativos con los casos ‘raros’ en los que las entradas no hacen lo que esperamos

Donde los testbenches no llegan

- Si hacemos un testbench de (por ejemplo) 10 M ciclos,
  - ¿cómo sabemos que no falla en el ciclo 10M + 1 ?
- Podemos probar que nuestro diseño es seguro?
  - Safety: asegurar que no se producen cuelgues y/o condiciones peligrosas para la salud, incluyendo cuelgues (ej: control de potencia, ingeniería biomédica, aviónica)
  - Security: asegurar que información específica no entra/sale del circuito (ej: criptografía, seguridad)
- Demostrar *propiedades* de nuestro circuito

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

¿Qué es la verificación formal?

El acto de probar o refutar un teorema, programa o algoritmo utilizando métodos matemáticos

¿Cómo se puede aplicar esto a circuitos digitales?

No hay testbench

- El usuario define:
  - Las propiedades del circuito
  - Las suposiciones al respecto de sus entradas
- Existe un ‘solver’ que intenta encontrar casos en los que no se cumplen las propiedades que debería tener el circuito
  - El solver es exhaustivo: ¡considera todas las posibilidades!
- Si encuentra un contraejemplo, genera una traza (`.vcd`) y un testbench (`.v`)

SMT solver?

- Un SMT (Satisfiability Modulo Theory) solver intenta satisfacer un teorema
  - En nuestro caso: “existe un conjunto de entradas que hagan fallar a nuestro circuito?”
- ¿Si no puede satisfacer el teorema?
  - Se demuestra que se cumplen las propiedades de nuestro circuito
- ¿Si puede satisfacerlo?
  - Genera una traza con las entradas que provocan el error

Limitaciones de la verificación formal

- No escala bien a circuitos con muchos registros
  - Crecimiento exponencial del nº de estados posibles
  - Siempre podemos trabajar con los submódulos de forma independiente
- No funciona bien con multiplicadores anchos
  - Muchos valores posibles
- Ya que se trata de hacer una demostración exhaustiva, el número de posibilidades totales puede ser un problema
  - Si son muchas, puede ser que la demostración nunca termine

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Propiedades y suposiciones

assert() , assume()

`assert(condition)`

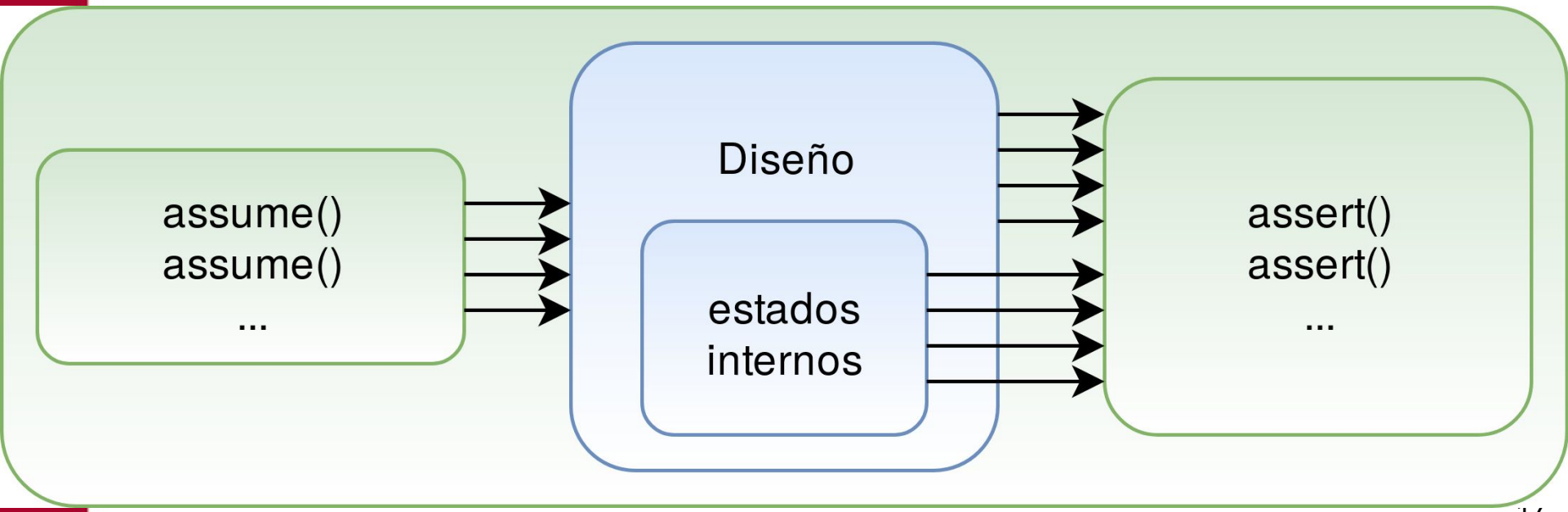
- Condiciones que deben cumplirse siempre
  - Las aplicaremos en las salidas y los estados internos

`assume(condition)`

- Suposiciones sobre el entorno en el que trabaja el circuito
  - Limitan el espacio de estados que debe explorar el solver
  - Las aplicaremos en las entradas

Propiedades y suposiciones

- Diseño en VHDL o Verilog
- `assume()` y `assert()` en Verilog o PSL
  - Nosotros usaremos PSL (soportado por yosys + ghdlsynth)



Propiedades y suposiciones

assert() , assume() de PSL

`assert(condition)`

- Condiciones que deben cumplirse siempre

```
assert always (Q <= MAX_COUNT);
```

`assume(condition)`

- Suposiciones sobre el entorno en el que trabaja el circuito

```
assume rst = '1';
```

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Property Specification Language

- PSL en código
  - Sólo en VHDL-2008 en adelante

```
assert always (Q <= MAX_COUNT);
```

- PSL en comentarios
  - Líneas deben empezar por -- psl

```
-- psl assert always (Q <= MAX_COUNT);
```

  - Hay que indicar al simulador que los lea

    (opción `-fpsl` en GHDL)

PSL: relaciones temporales

- PSL, además de assertions típicos, permite expresar relaciones temporales:
  - `assert always (a |-> next(b));`
    - Si **a** se cumple, al siguiente ciclo de reloj se ha de cumplir **b**

- Ejemplos:

```
assert a;  -- a must be true in the first clock cycle
assert always a;  -- a must be true for all clock cycles
assert never (a and b);  -- a and b cannot be active at the same time
```

PSL

- Los assert en PSL también pueden incluir un report
- Etiqueta en un assert:
  - Si la indicamos, la herramienta se referirá al assert por la etiqueta (por ejemplo cuando falle)
  - Si no, se referirá al assertion indicando fichero y línea
- No todo PSL está soportado por las herramientas libres!
  - En PSL examples for GHDL and symbiosys viene indicado todo el 'subset' que está soportado
    - 'synthesis only' significa que sólo vale para verificación formal, pero no para simulación
    - Para poder aplicar el solver debe hacerse la síntesis antes -> sólo podemos verificar formalmente código que sea sintetizable
- Podemos usar PSL para verificación funcional también
  - Tiene un potencial enorme para hacer, por ejemplo, monitores de protocolo

PSL: Sintaxis

- Sintaxis:
  - [etiqueta:] directiva [propiedad]
- Directivas:
  - assert, assume, cover, restrict
- Propiedades:
  - Pueden incluir valores de objetos (signals, ports) y operadores temporales
- Operadores temporales:
  - always, never, -> (implicación), <-> (if and only if), next, next[n], prev, prev[n]
- Expresiones secuenciales:
  - SERE - Sequential Extended Regular Expressions
  - : (fusión), ; (concatenación), |-> (implicación solapante), |=> (implicación no solapante), [*], [+], [*i], [*i to j] (repetición consecutiva), [=i], [=i to j] (repetición no consecutiva)
- Y más…
  - Pero con esto tenemos para hacer muchísimas cosas!

PSL: ejemplos

- Primeros pasos:

```
assert a;  -- a must be true in the first clock cycle
assert always a;  -- a must be true for all clock cycles
assert never (a and b);  -- a and b cannot be active at the same time
```

- Implicaciones:

```
assert always (a |-> b);  -- if a, then b must be true at the same time
assert always (a |-> next(b));  -- if a, then b must be true the next clock cycle
assert always (a |=> b);  -- if a, then b must true the next clock cycle
```

- Secuencias temporales:

```
assert always {a;b} |-> {c};  -- if a is true and in the next cycle b is true, then c must be true in the same cycle when b is true
assert always {a;b} |-> {c;d};  -- if the sequence {a;b} happens, the secuence {c;d} must follow, with c overlapping b
assert always {a;b} |=> {c;d};  -- if the sequence {a;b} happens, the secuence {c;d} must follow, with c being true one cycle after b
```

PSL: secuencias

- Fusión (:) es a concatenación (;) lo que overlapping implication (|->) es a non-overlapping implication (|=>)
  - **{a;b}** b empieza al ciclo siguiente que a termine
  - **{a:b}** b empieza en el ciclo en que a termina
- Al final son todo secuencias!
  - {secuencia1} |-> {secuencia2}
- Tiene sentido pensar en regex (regular expressions):
  - **a[*]** 'matchea' ninguna, una o más repeticiones consecutivas de a
  - **a[+]** 'matchea' una o más repeticiones consecutivas de a
  - **a[*3]** 'matchea' 3 repeticiones consecutivas de a
  - **a[*2 to 4]** 'matchea' entre 2 y 4 repeticiones consecutivas de a
  - **a[=2]** 'matchea' 2 repeticiones (no necesariamente consecutivas) de a
  - **a[=3 to 6]** 'matchea' entre 3 y 6 repeticiones (no necesariamente consecutivas) de a
- Puedes dibujar o describir las siguientes secuencias y relaciones entre ellas?
  - {a;b} |-> {c;d}
  - {a;b} |=> {c;d}
  - { a[*2 to 3] ; b[=2] } |=> { c : d[=1 to 3] }

¿Cómo expresar propiedades que dependen de valores anteriores?

Lo mejor es expresarlas usando secuencias, pero si no es posible:

`prev(signal)` → valor en el ciclo anterior

`prev(signal, N)` → valor hace N ciclos

Ojo! Si le pedimos el valor de una señal antes de tiempo 0, ¡puede salir cualquier cosa!

- Los `assume()` se cumplirán
- Los `assert()` seguramente no!

Antes de usar `prev(x)` o `prev(x,N)` tenemos que asegurarnos de que han pasado N ciclos

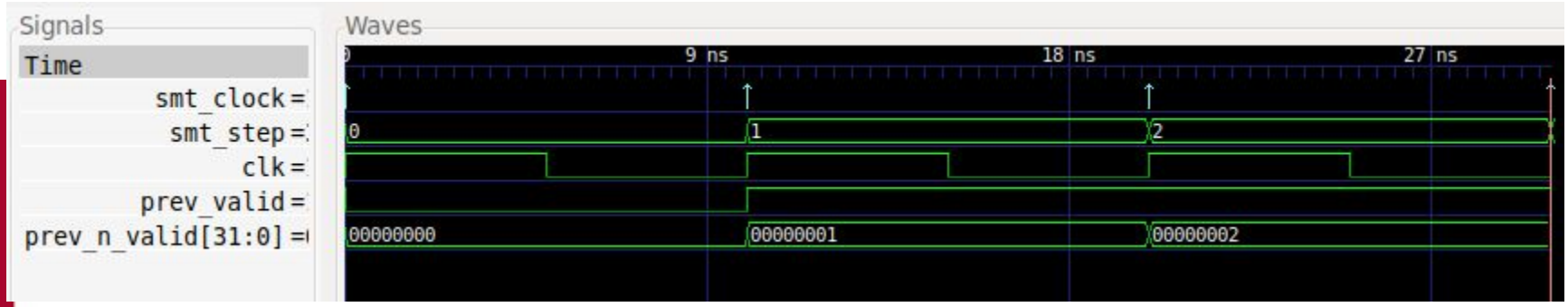
Actualmente (2021) `prev()` es 'synthesis-only' (no simulable) en GHDL

¿Cuándo podemos utilizar `prev()`?

Una solución es definir un signal que indique si es válido:

```
[...]

   signal prev_valid : boolean := false;
   signal prev_N_valid : integer := 0;

begin
   default clock is rising_edge(clk);
```

```
process(clk)
begin
  if rising_edge(clk) then
    prev_valid <= true;
    prev_N_valid <= prev_N_valid + 1;
  end if;
end process;
```



Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Chequeo de modelo acotado (BMC)

Comprobar que nuestras propiedades se cumplen durante los N primeros ciclos

El solver empieza en tiempo 0 y explora exhaustivamente N ciclos

- El circuito empieza en un estado conocido (si tenemos bloques valores iniciales y/o reset)

Prueba matemáticamente que nuestro circuito funciona correctamente durante N ciclos

Ejemplo

Contador de 8 bits

Diseño (VHDL)

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  generic ( MAX_COUNT : integer := 100);
  port ( clk, rst: in std_logic;
         Q: out unsigned(7 downto 0));
end counter;

architecture behavioral of counter is
  signal count: unsigned(7 downto 0);
  signal p_count: unsigned(7 downto 0);
begin

sinc: process(clk, rst)
    begin
        if (rst='1') then
            count <= (others=>'0');
        elsif (rising_edge(clk)) then
            count <= p_count;
        end if;
    end process;

comb: process(count)
    begin
        if (count = MAX_COUNT) then
          p_count <= (others => '0');
        else
          p_count <= count + 1;
        end if;
    end process;

    Q <= count;

end behavioral;
```

Propiedades (PSL)

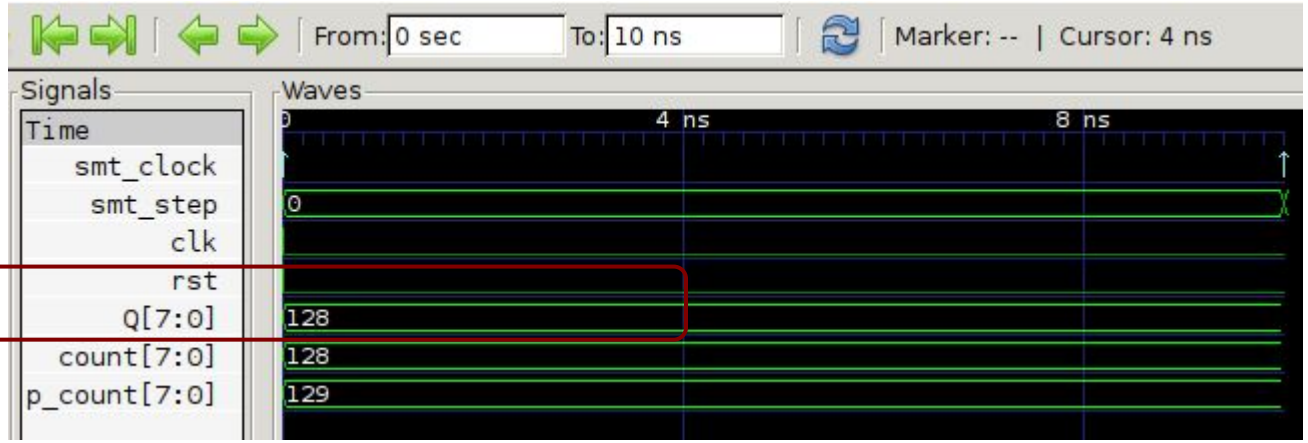
```vhdl
-- Default clock for PSL assertions
default clock is rising_edge(clk);

-- PSL assertions
assert always (Q <= MAX_COUNT);
```

Bounded Model Checking

Ejemplo

```
[options]
mode bmc
depth 200
```

```
$ sby --yosys "yosys -m ghdl" -f counter.sby
[...]
Checking assumptions in step 0..
Checking assertions in step 0..
BMC failed!
Assert failed in counter: /15
Writing trace to VCD file: engine_0/trace.vcd
Writing trace to Verilog testbench: engine_0/trace_tb.v
```



Propiedades (PSL)

```
-- Default clock for PSL assertions
default clock is rising_edge(clk);

-- PSL assertions
assert always (Q <= MAX_COUNT);
```

Propiedades (PSL)

```
-- Default clock for PSL assertions
default clock is rising_edge(clk);

-- PSL assertions
assert always (Q <= MAX_COUNT)

-- Force a reset in the first clock cycle
-- since we don't have an 'always', this assumption only applies to the first clock cycle
assume rst = '1';
```

Ejemplo

```
$ sby --yosys "yosys -m ghdl" -f counter.sby
[...]
Checking assumptions in step 0..
Checking assertions in step 0..
Checking assumptions in step 1..
Checking assertions in step 1..
[...]
Checking assumptions in step 198..
Checking assertions in step 198..
Checking assumptions in step 199..
Checking assertions in step 199..
Status: PASSED
```

```
[options]
mode bmc
depth 200
```

Como se cumplen las propiedades, no genera ninguna traza

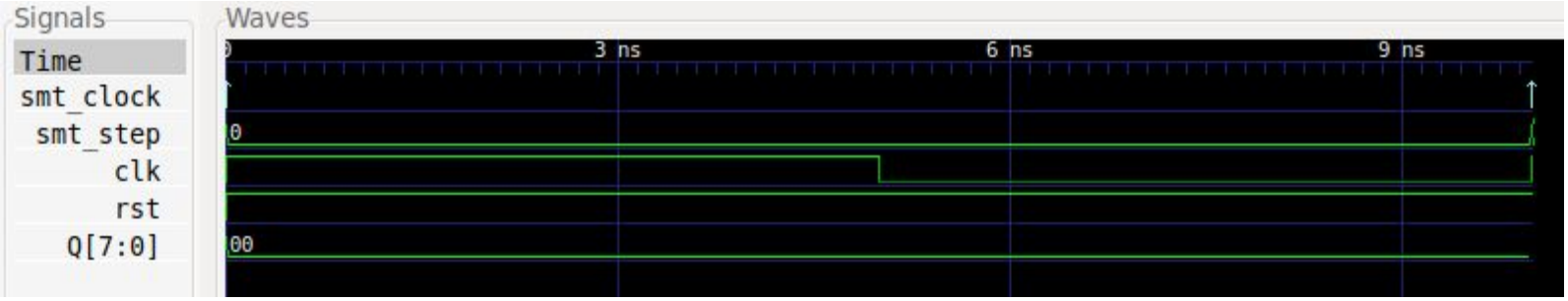
Ejemplo

¿Qué ocurre si añadimos una nueva propiedad?

```
-- Assert the counter is always counting up
assert always (Q-prev(Q) = 1);
```

```
Checking assertions in step 0..
BMC failed!
```

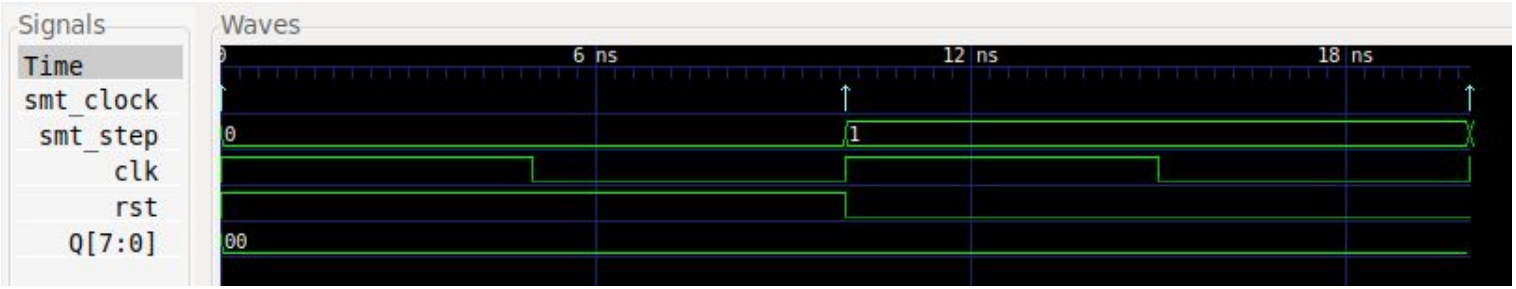
no podemos usar prev() en tiempo cero!



Ejemplo

```
-- Assert the counter is always counting up, except for the first clock cycle
assert always (not rst = '1') -> (Q-prev(Q) = 1);
```

```
Checking assertions in step 1..
BMC failed!
```

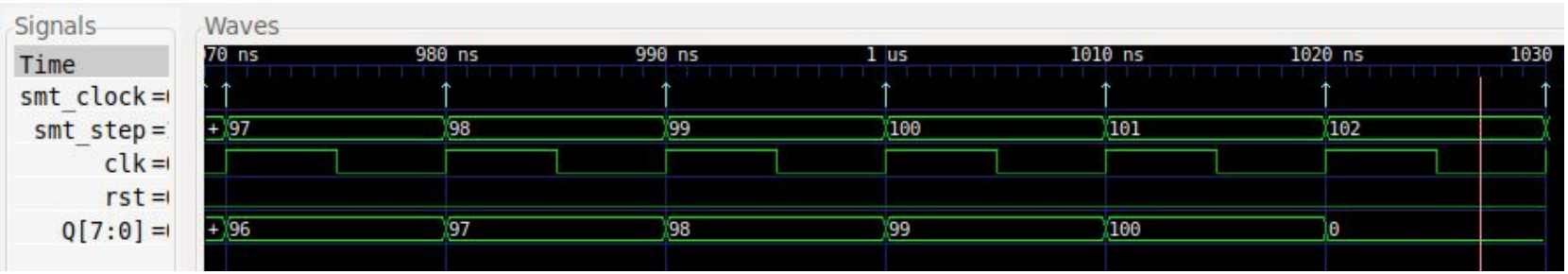


Bounded Model Checking

Ejemplo

```
-- Assert the counter is always counting up, excepting cycle zero
and if it was reset on last cycle
assert always (not rst = '1') and (not prev(rst) = '1') ->
  (Q-prev(Q) = 1);
```

```
Checking assertions in step 102..
BMC failed!
```



Bounded Model Checking

```
-- Assert the counter is always counting up, but only:
 -- 1) If it's not the first clock cycle
 -- 2) If it hasn't just been reset
 -- 3) If last value wasn't MAX_COUNT
 -- assert always (Q /= MAX_COUNT) and not rst;
 assert always (not (rst = '1') and (not prev(rst) = '1') and
(prev(Q) /= MAX_COUNT)) -> (Q-prev(Q) = 1);
```

```
Checking assertions in step 199..
Status: PASSED
```

Limitaciones

Demuestra corrección del ciclo 0 al N-1

El circuito podría fallar justo en el ciclo N !

Podríamos incrementar N y cambiarlo por M (con M > N)

… pero el circuito podría fallar en el ciclo M !

Existe alguna forma de generalizar estos resultados?

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Inducción matemática

Se puede demostrar que la propiedad P(n) se cumple para todo N >= 0 si se cumplen:

- Se cumple P(0)
- Dado P(k), puede demostrarse que P(k+1)

Inducción matemática

Se puede demostrar que la propiedad P(n) se cumple para todo N >= 0 si se cumplen:

- P(0)
- $P(k) \rightarrow P(k+1)$

Inducción matemática

Se puede demostrar que la propiedad P(n) se cumple para todo N >= 0 si se cumplen:

- P(0)
  - (La propiedad se cumple en N = 0)
- P(k) → P(k+1)

Inducción matemática

Se puede demostrar que la propiedad P(n) se cumple para todo N >= 0 si se cumplen:

- P(0)
  - (La propiedad se cumple en N = 0)
- P(k) → P(k+1)
  - (Dado P(k), puede demostrarse que P(k+1))

Inducción matemática

Se puede demostrar que la propiedad P(n) se cumple para todo N >= 0 si se cumplen:

- P(0)
  - (La propiedad se cumple en N = 0)
- P(k) → P(k+1)
  - (Dado P(k), puede demostrarse que P(k+1))
  - (Dicho de otra forma: si se cumple para un valor, esto implica que se cumplirá para el siguiente)

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - 0 = 0 * (0 + 1)

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - 0 = 0 * (0 + 1) Sí

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - 0 = 0 * (0 + 1) Sí
- Suponiendo que se cumple en n = k:

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - 0 = 0 * (0 + 1) Sí
- Suponiendo que se cumple en n = k:
  - $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - $0 = 0 * (0 + 1)$ Sí
- Suponiendo que se cumple en n = k:
  - $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- ¿Podemos demostrar que se cumple en n = k+1 ?

Ejemplo

Demostramos

$$0 + 2 + 4 + \ldots + 2n = n(n + 1)$$

- Se cumple en n = 0 ?
  - $0 = 0 * (0 + 1)$ Sí
- Suponiendo que se cumple en n = k:
  - $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- ¿Podemos demostrar que se cumple en n = k+1 ?
  - $0 + 2 + 4 + \ldots + 2k + 2(k+1) = (k+1)(k+2)$ eq. b)

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- Sumamos $2(k+1)$ a ambos lados:

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- Sumamos $2(k+1)$ a ambos lados:
- $0 + 2 + 4 + \ldots + 2k + \mathbf{2(k+1)} = k(k+1) + \mathbf{2(k+1)}$

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- Sumamos $2(k+1)$ a ambos lados:
- $0 + 2 + 4 + \ldots + 2k + 2(k+1) = k(k+1) + 2(k+1)$
- Reordenando el lado derecho:

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- Sumamos $2(k+1)$ a ambos lados:
- $0 + 2 + 4 + \ldots + 2k + 2(k+1) = k(k+1) + 2(k+1)$
- Reordenando el lado derecho:
- $0 + 2 + 4 + \ldots + 2k + 2(k+1) = (k+1)(k+2)$ eq. b)

Ejemplo

Partiendo de eq. a), ¿podemos llegar a eq. b) ?

- $0 + 2 + 4 + \ldots + 2k = k(k+1)$ eq. a)
- Sumamos $2(k+1)$ a ambos lados:
- $0 + 2 + 4 + \ldots + 2k + 2(k+1) = k(k+1) + 2(k+1)$
- Reordenando el lado derecho:
- $0 + 2 + 4 + \ldots + 2k + 2(k+1) = (k+1)(k+2)$ eq. b)

Esto (junto con P(0)) demuestra que la propiedad se cumple para N >= 0

En circuitos digitales

Esto se puede utilizar para generalizar los resultados del Bounded Model Checking:

- Demostramos que las propiedades de nuestro circuito se cumplen en tiempo 0
- El solver demuestra que si se cumplen en tiempo k, se cumplirán en k + 1

Pero...

En circuitos digitales

El problema es que existen propiedades de los circuitos que necesitan más de 1 ciclo para saber si la evolución es correcta

(Ej: 12 ciclos después de `req` se activa `ack`)

¿Cómo se puede hacer la demostración?

1 ciclo

- Demostramos que las propiedades se cumplen en tiempo 0
  - (BMC con N = 1)
- El solver demuestra que si se cumplen en tiempo k, se cumplirán en k + 1
  - (k-induction con N = 1)

2 ciclos

- Demostramos que las propiedades se cumplen en tiempo 0 y 1
  - (BMC con N = 2)
- El solver demuestra que si se cumplen en tiempo k-1 y k, se cumplirán en k + 1
  - (k-induction con N = 2)

3 ciclos

- Demostramos que las propiedades se cumplen en tiempo 0, 1 y 2
  - (BMC con N = 3)
- El solver demuestra que si se cumplen en tiempo k-2, k-1 y k, se cumplirán en k + 1
  - (k-induction con N = 3)

4 ciclos

- Demostramos que las propiedades se cumplen en tiempo 0, 1, 2, 3
  - (BMC con N = 4)
- El solver demuestra que si se cumplen en tiempo k-3, k-2, k-1, k, se cumplirán en k + 1
  - (k-induction con N = 3)

N ciclos

- Demostramos que las propiedades se cumplen en tiempo 0 .. N-1
  - (BMC con profundidad N)
- El solver demuestra que si se cumplen en tiempo k-(N-1) .. k se cumplirán en k + 1
  - (k-induction con profundidad N)

k-(N-1) .. k ?

El solver pone a nuestro circuito en el estado que quiera

- Sus únicas limitaciones son los `assume()` y `assert()` que hayamos puesto en las propiedades ...
- … y NO lo que a nosotros nos podría parecer la ‘evolución natural’ del circuito

Ej: dos submódulos iguales, con las mismas entradas, pueden “empezar” con estados internos diferentes!

k-(N-1) .. k ?

Ej: dos submódulos iguales, con las mismas entradas, pueden “empezar” con estados internos diferentes!

¿Cómo podemos arreglar esto?

- Mejorar suposiciones sobre las entradas y/o incrementar N de forma que dé tiempo a que se propaguen las entradas por ambos módulos
- Añadir propiedades para garantizar que los estados internos de los submódulos siempre son iguales

¿Cómo se utiliza?

- Primero describimos las propiedades necesarias para pasar BMC
- Luego posiblemente tengamos que incluir más propiedades para que pase k-induction
- Cuando pase k-induction, sabremos que las propiedades se cumplen para cualquier ciclo de reloj >= 0

Ejemplo 1

Contador de 8 bits

```
[options]
mode prove
depth 200
```

```
engine_0.induction: ##   0:00:00  Trying induction in step 200..
engine_0.induction: ##   0:00:00  Trying induction in step 199..
engine_0.induction: ##   0:00:00  Temporal induction successful.
engine_0.induction: ##   0:00:00  Status: passed
engine_0.induction: finished (returncode=0)
engine_0: Status returned by engine for induction:  pass
```

Ejemplo 1

Contador de 8 bits

```
[options]
mode prove
depth 200
```

```
engine_0.basecase: ##   0:00:00  Checking assumptions in step 0..
engine_0.basecase: ##   0:00:00  Checking assertions in step 0..
[...]
engine_0.basecase: ##   0:00:13  Checking assumptions in step 199..
engine_0.basecase: ##   0:00:13  Checking assertions in step 199..
engine_0.basecase: ##   0:00:13  Status: passed
engine_0.basecase: finished (returncode=0)
engine_0: Status returned by engine for basecase:  pass
```

```
summary: engine_0 (smtbmc) returned pass for induction
summary: engine_0 (smtbmc) returned pass for basecase
summary: successful proof by k-induction.
DONE (PASS, rc=0)
```

Ejemplo 2

Contador de 8 bits

```
[options]
mode prove
depth 20
```

```
engine_0.induction: ##   0:00:00  Trying induction in step 20..
engine_0.induction: ##   0:00:00  Trying induction in step 19..
engine_0.induction: ##   0:00:00  Temporal induction successful.
engine_0.induction: ##   0:00:00  Status: passed
engine_0.induction: finished (returncode=0)
engine_0: Status returned by engine for induction: pass
```

Ejemplo 2

Contador de 8 bits

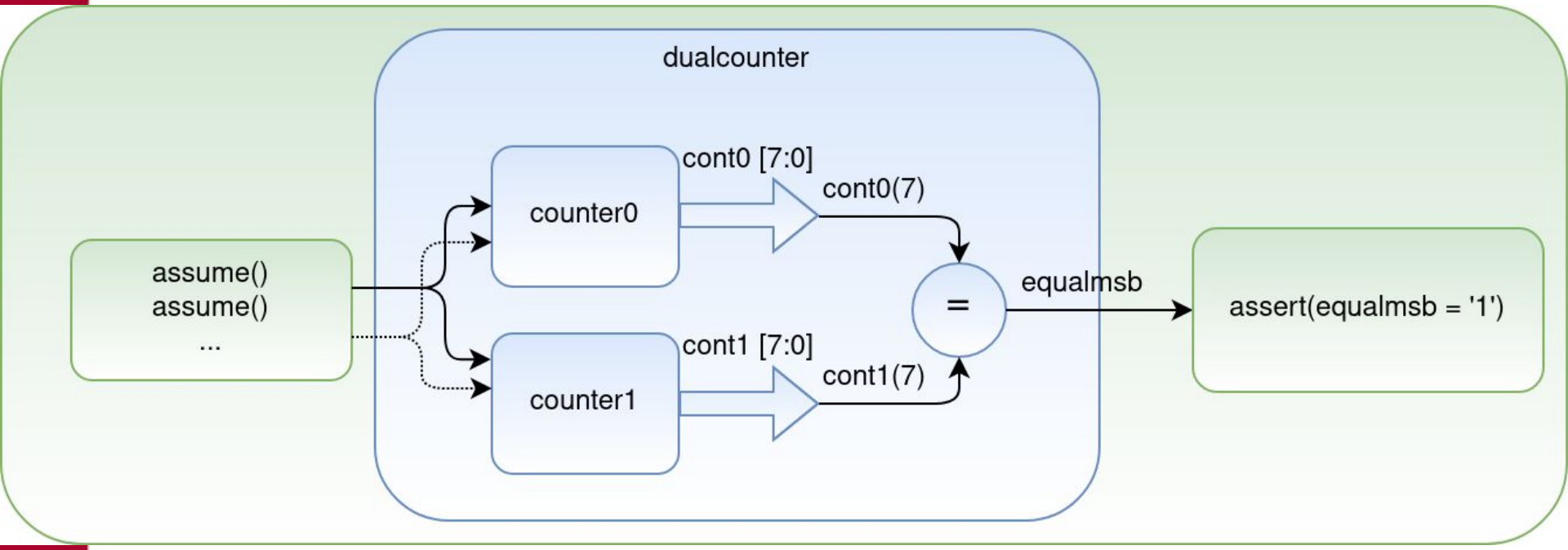
```
[options]
mode prove
depth 20
```

```
engine_0.basecase: ##   0:00:00  Checking assumptions in step 0..
engine_0.basecase: ##   0:00:00  Checking assertions in step 0..
[...]
engine_0.basecase: ##   0:00:13  Checking assumptions in step 19..
engine_0.basecase: ##   0:00:13  Checking assertions in step 19..
engine_0.basecase: ##   0:00:13  Status: passed
engine_0.basecase: finished (returncode=0)
engine_0: Status returned by engine for basecase: pass
```

```
summary: engine_0 (smtbmc) returned pass for induction
summary: engine_0 (smtbmc) returned pass for basecase
summary: successful proof by k-induction.
DONE (PASS, rc=0)
```

Ejemplo 3

2 contadores de 8 bits



Diseño (VHDL)

```
[..]

architecture Behavioral of dualcounter is
  signal count0: unsigned(7 downto 0);
  signal count1: unsigned(7 downto 0);
begin

  counter0: entity work.counter
  generic map (MAX_COUNT => MAX_COUNT)
  port map( rst => rst,
            clk => clk,
            Q   => count0);

  counter1: entity work.counter
  generic map (MAX_COUNT => MAX_COUNT)
  port map( rst => rst,
            clk => clk,
            Q   => count1);

  equalmsb <= '1' when (count0(7) =
                  count1(7)) else '0';

end Behavioral;
```

Propiedades (PSL)

```
-- Default clock
default clock is rising_edge(clk);
-- Assume a reset on the first clock
cycle
assume rst = '1';
-- Output equalmsb should always be
'1'
assert always equalmsb = '1';
```

Ejemplo 3

```
[options]
mode prove
depth 20
```

```
[...]
engine_0.induction: ##   0:00:00  Trying induction in step 0..
engine_0.induction: ##   0:00:00  Temporal induction failed!
engine_0.induction: ##   0:00:00  Assert failed in dualcounter: /33
engine_0.induction: ##   0:00:00  Writing trace to VCD file:
engine_0/trace_induct.vcd
[...]
Status returned by engine for induction: FAIL
[...]
engine_0.basecase: ##   0:00:00  Checking assertions in step 19..
engine_0.basecase: ##   0:00:00  Status: passed
engine_0: Status returned by engine for basecase: pass
engine_0 (smtbmc) returned FAIL for induction
engine_0 (smtbmc) returned pass for basecase
```

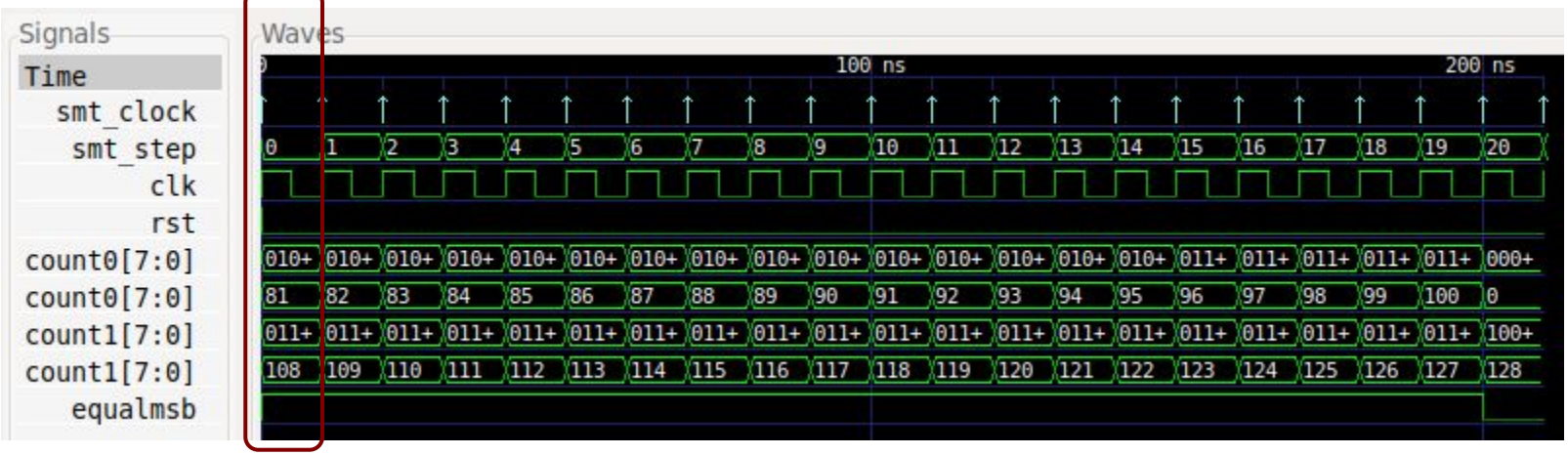
k-induction

Ejemplo 3

```
[options]
mode prove
depth 20
```

Propiedades (PSL)

```
-- Output equalmsb should always be
'1'
assert always equalmsb = '1';
```



Dos posibles soluciones

a)

```
[options]
mode prove
depth 200
```

basecase: PASS

induction: PASS

b)

Propiedades (Verilog)

```
-- Assert that equalmsb is always '1'
assert always equalmsb = '1';
-- Assert that both counter outputs are always equal
assert always count0 = count1;
```

basecase: PASS

induction: PASS

Dos posibles soluciones

a)

```
[options]
mode prove
depth 128
```

basecase: PASS

induction: PASS

a???)

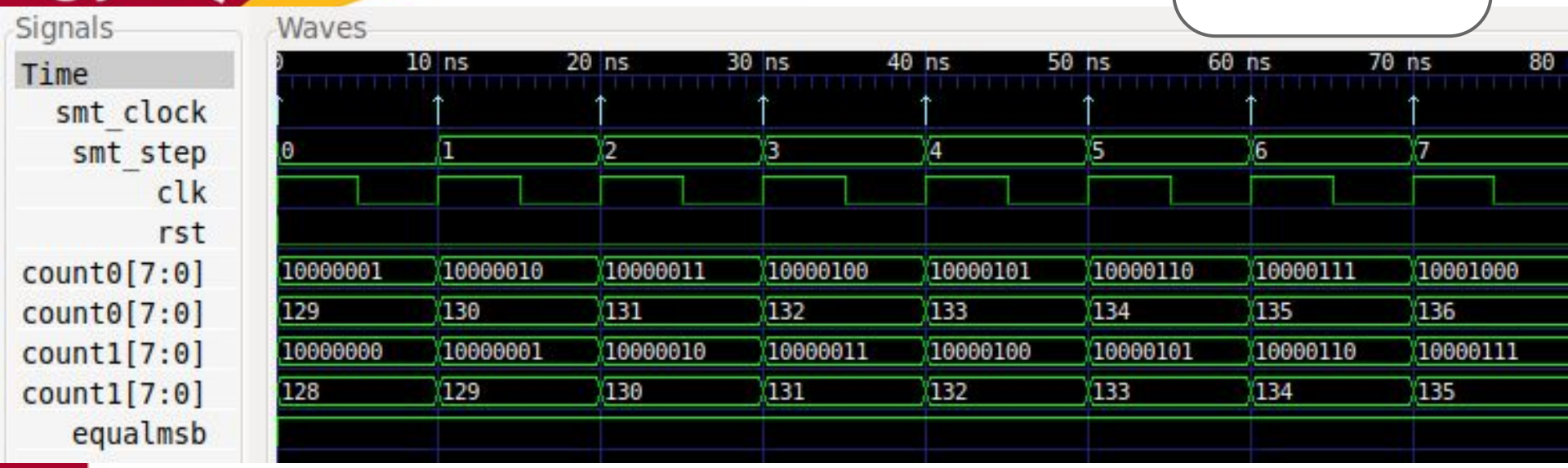
```
[options]
mode prove
depth 127
```

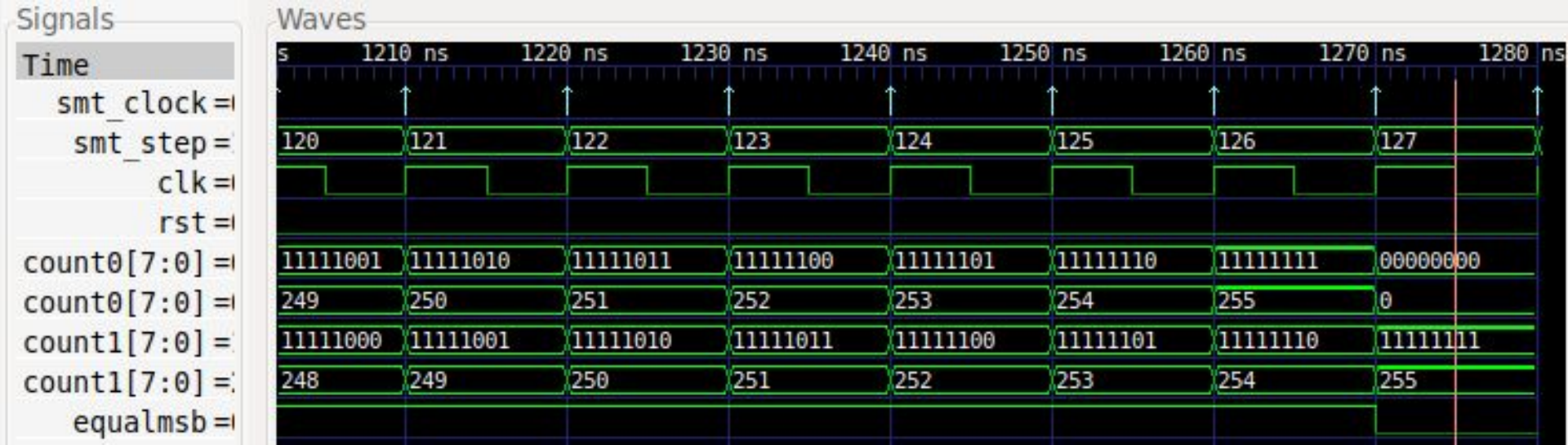
basecase: PASS

induction: FAIL

k-induction

```
[options]
mode prove
depth 127
```





Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

¿Y si todo funciona pero quiero generar una traza?

Por defecto no se generará traza si no falla ningún assertion

Podemos usar `cover(condition)`

Indicaremos a la herramienta que funcione en modo cover

Ejemplo

Contador de 8 bits

```
[options]
mode cover
depth 200
```

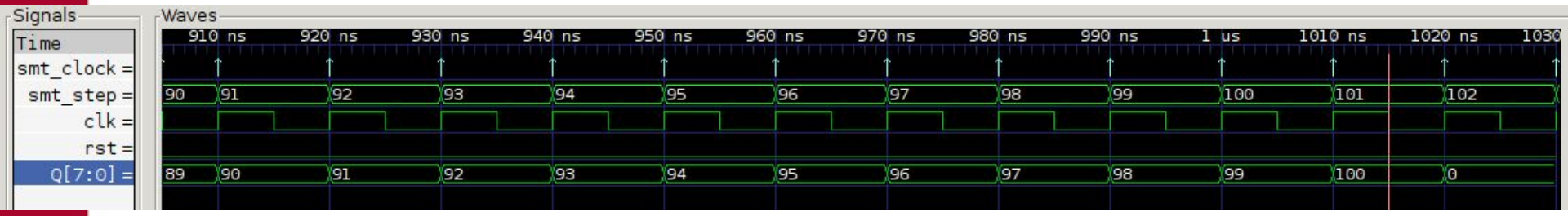
```
-- Cover the overflow -> zero case
-- What we write here has to be a sequence inside curly braces {}, even if
-- we only have one element (for example, { Q = MAX_COUNT } )
cover {Q = MAX_COUNT ; Q = 0};
```

Ejemplo

Contador de 8 bits

```
[options]
mode cover
depth 200
```

```
Checking cover reachability in step 0..
Checking cover reachability in step 1..
[...]
Checking cover reachability in step 101..
Checking cover reachability in step 102..
Reached cover statement at counter_properties.sv:61 in step 102.
Writing trace to VCD file: engine_0/trace0.vcd
Writing trace to Verilog testbench: engine_0/trace0_tb.v
Writing trace to constraints file: engine_0/trace0.smtc
Status: passed
```



Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

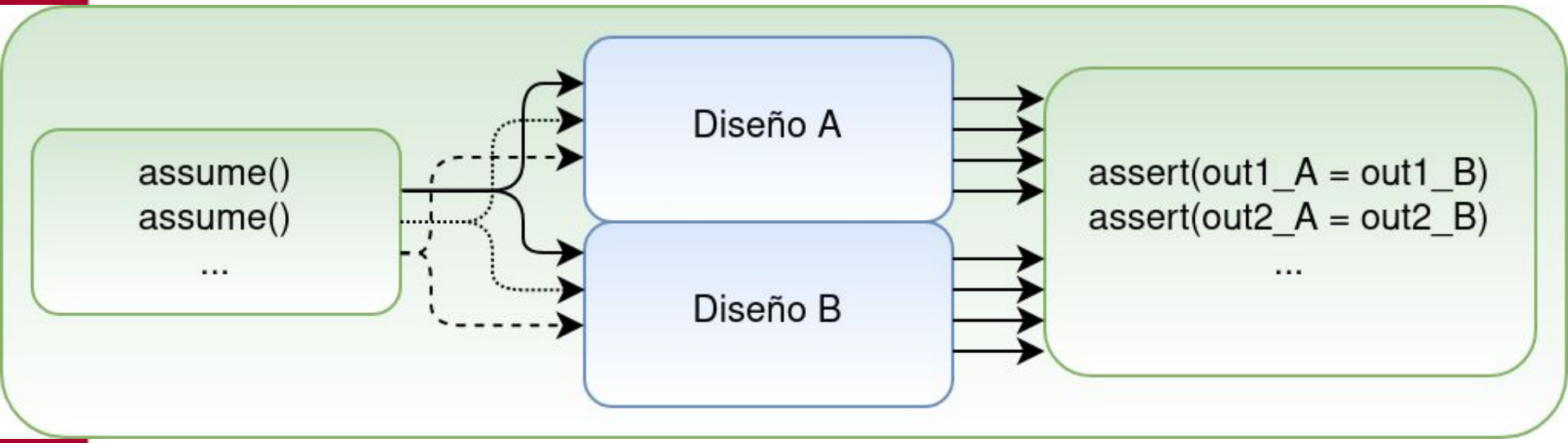
Comprobar que dos diseños se comportan igual

Utilizado:

- Para comprobar si un sintetizador genera circuitos correctos
- Para comprobar que una transformación sobre un circuito no ha roto la funcionalidad
- Tras una reestructuración de código
- Al traducir un módulo de Verilog a VHDL o viceversa

Equivalence checking

¿Podemos demostrar que las salidas son iguales para cualquier ciclo de reloj?



Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Conclusiones

- La verificación formal es otra forma que tenemos de incrementar la confianza en nuestro diseño
  - Puede usarse de manera complementaria a la verificación funcional
  - Es especialmente buena encontrando situaciones que no habíamos previsto
- Funciona bien en módulos pequeños/medianos y sin multiplicadores de gran anchura
  - Para módulos de procesado de señal: podemos aplicarla a señales de control y estados internos aunque no la apliquemos a los datos

Conclusiones

- BMC sólo demuestra cumplimiento de las propiedades durante los primeros N ciclos
- k-induction necesita y complementa al BMC
  - Con ambos, demuestras que la(s) propiedad(es) se cumple(n) *siempre*
  - Esto no se puede hacer con verificación funcional! (salvo en casos muy concretos)
- Verificación formal es white-box
  - Verificación funcional es black-box

Conclusiones

- Los signals que añadamos a las propiedades (prev_valid u otros) deben ser sintetizables
  - Por ejemplo, no se pueden utilizar números reales
- Todos los bugs que se pueden encontrar con verificación formal se pueden encontrar también con verificación funcional
  - Pero el solver es más ‘creativo’ buscando problemas

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Introducción a PSL
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Bibliografía

- C. Eisner, D. Fisman, A Practical Introduction to PSL
  - Muy completo y bien explicado
- T. Meissner, PSL examples for GHDL and symbiosys
  - Indica qué subset está soportado por las herramientas libres y contiene muchos ejemplos
- YosysHQ, sby file format reference
- Gisselquist, Dan, An Introduction to Formal Methods
  - Bien explicado, aunque aquí los assume y assert están en Verilog
- Wikipedia, the Free Encyclopedia: Property Specification Language
  - Contiene ejemplos sencillos bien explicados

Bibliografía

- Seligman, E., Schubert, T., Achutha Kiran Kumar, M. V., *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Elsevier, 2015
- Brinkmann, R., Kelf, D., “Formal Verification—The Industrial Perspective”, en *Formal System Verification: State-of-the-Art and Future Trends*, R. Drechsler, ed., Springer, 2018, cap. 5, págs. 155-182

Resultados de aprendizaje

- Conocer las limitaciones de los testbenches clásicos
- Conocer el potencial y las limitaciones de los métodos de verificación formal
- ¿Cómo se describen las propiedades de y suposiciones sobre un circuito?
- ¿Para qué sirve `prev()` y cuándo podemos utilizarla?
- ¿En qué consiste el Bounded Model Checking? ¿Qué demuestra sobre un circuito?
- ¿Qué es el k-induction? ¿Qué demuestra sobre un circuito?
- ¿Cuándo interesa hacer un chequeo de equivalencia entre circuitos?