Tema 7: Métodos de Verificación Formal para Circuitos Digitales

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

Acknowledgement to Dan Gisselquist,
Symbiotic EDA

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 y Verilog
- 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
- Propiedades multi-ciclo
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Propiedades multi-ciclo
- 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
- Propiedades multi-ciclo
- 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
  - 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
- Propiedades multi-ciclo
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

assert() , assume() de Verilog

`assert(condition)`

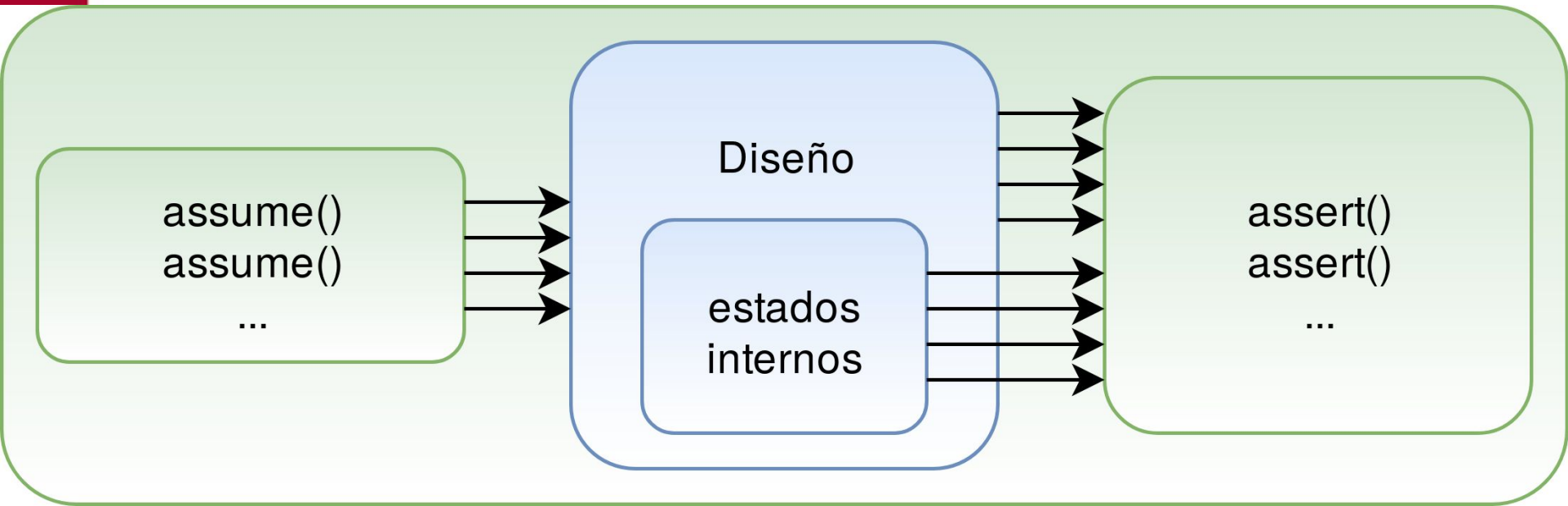
- Condiciones que deben cumplirse siempre
  - Las aplicaremos en las salidas y los estados internos

`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
  - Usando sby (SymbiYosys): Verilog
  - Soporte para PSL planeado para el futuro (a través de ghdlsynth)



assert() , assume() de Verilog

`assert(condition)`

- Condiciones que deben cumplirse siempre

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

`assume(condition)`

- Suposiciones sobre el entorno en el que trabaja el circuito

```
always @(*)
  if (f_past_valid == 1'b0)
    assume(rst == 1'b1);
```

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Propiedades multi-ciclo
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Propiedades multi-ciclo

¿Cómo expresar propiedades que dependen de valores anteriores?

`$past(signal)` → valor en el ciclo anterior

`$past(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!

Propiedades multi-ciclo

¿Cuándo podemos utilizar $past()?

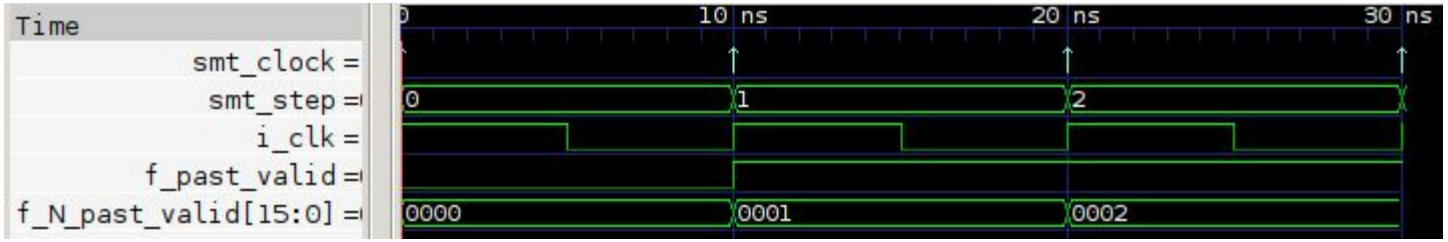
Una solución es definir un reg que indique si es válida:

```
reg f_past_valid;

initial f_past_valid = 1'b0;
always @(posedge clk)
  f_past_valid <= 1'b1;
```

```
reg [15:0] f_N_past_valid;

initial f_N_past_valid = 16'b0;
always @(posedge clk)
        f_N_past_valid =
                f_N_past_valid + 1;
```



Ejemplo

`ack` debe activarse 12 ciclos después de `req`

```
always @(posedge(clk))
begin
    if (f_N_past_valid >= 12)
        if ($past(req, 12))
            assert (ack);
```

`$past` sólo puede utilizarse dentro de un `always @(posedge(...))`

Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Propiedades multi-ciclo
- 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 initial (verilog) 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;
```

```vhdl
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;
```

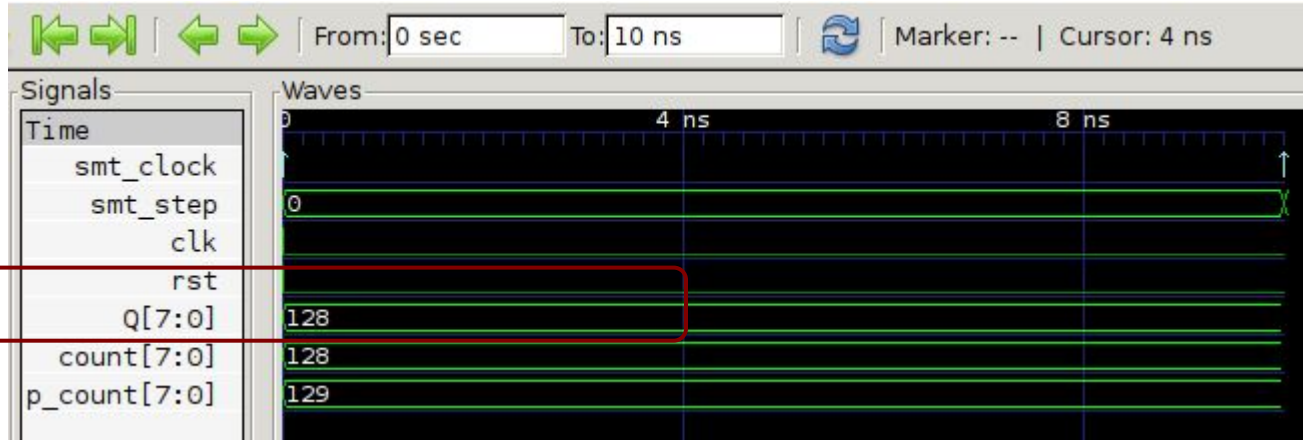
Propiedades (Verilog)

```
`ifdef FORMAL
   always @(*)
     assert (Q <= MAX_COUNT);
`endif
```

Ejemplo

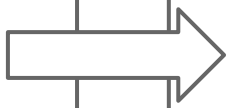
```
[options]
mode bmc
depth 200
```

```
$ sby -f counter.sby
[...]
Checking assumptions in step 0..
Checking assertions in step 0..
BMC failed!
Assert failed in counter.copy: counter_properties.sv:30
Writing trace to VCD file: engine_0/trace.vcd
Writing trace to Verilog testbench: engine_0/trace_tb.v
```



Propiedades (Verilog)

```
`ifdef FORMAL
      always @(*)
            assert (Q <=
MAX_COUNT);
`endif
```



Propiedades (Verilog)

```
`ifdef FORMAL
    reg f_past_valid;

    initial
      f_past_valid <= 1'b0;

    always @(posedge clk)
      f_past_valid <= 1'b1;

    // Force a reset in the
    // first clock cycle
    always @(*)
      if (f_past_valid == 1'b0)
        assume(rst == 1'b1);

    always @(*)
      assert (Q <= MAX_COUNT);
`endif
```

Ejemplo

```
$ sby -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
```

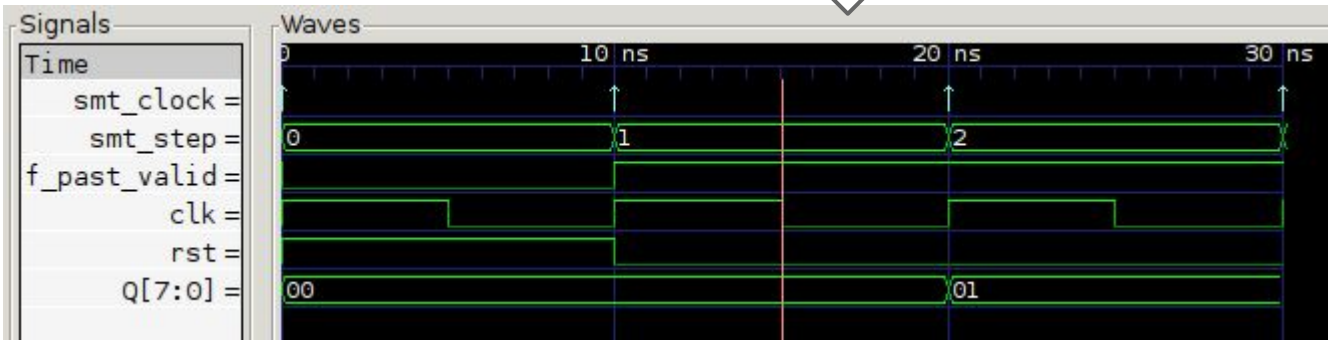
Como se cumplen las propiedades, no genera ninguna traza

Ejemplo

¿Qué ocurre si añadimos una nueva propiedad?

```
// Assert the counter is always counting up
always @(posedge clk)
    if (f_past_valid)
        assert (Q - $past(Q) == 1);
```

```
Checking assertions in step 2..
BMC failed!
```

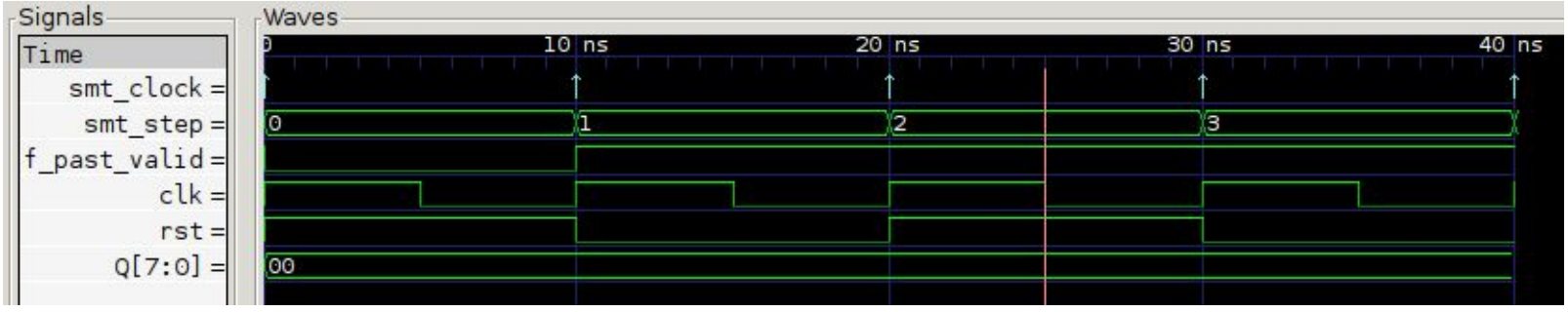


Bounded Model Checking

Ejemplo

```
// Assert the counter is always counting up, unless it has been
reset
        always @(posedge clk)
        begin
          if (f_past_valid)
            if ($past(rst) == 1'b0)
              assert (Q - $past(Q) == 1);
        end
```

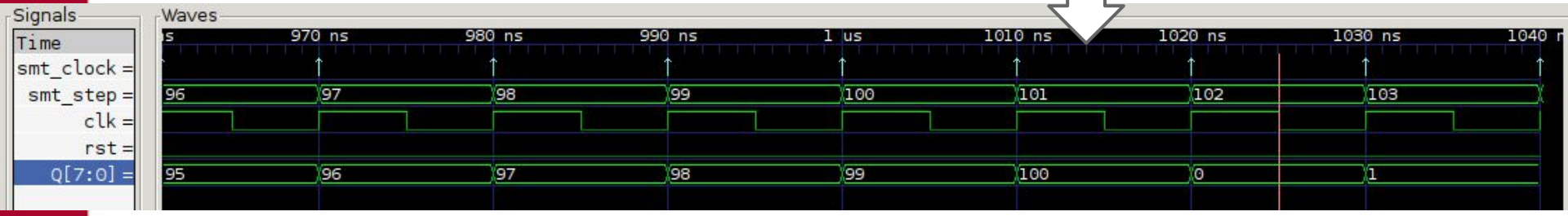
```
Checking assertions in step 3..
BMC failed!
```



```
// Assert the counter is always counting up:
// 1) If we have been reset, just check that Q = 0
// 2) If we haven't been reset, check we are counting up
always @(posedge clk)
begin
  if (f_past_valid)
  begin
    if (rst == 1'b1)
      assert (Q == 1'b0);
    else if ($past(rst) == 1'b0)
      assert (Q - $past(Q) == 1);
    end
end
```

Checking assertions in step 103..

BMC failed!



```
// Assert the counter is always counting up but keep in mind
// wraparound at MAX_COUNT:
// 1) If we have been reset, just check that Q = 0
// 2) If we haven't been reset, check we are counting up
// 3) If last value was MAX_COUNT, current value should be 0
        always @(posedge clk)
        begin
          if (f_past_valid)
          begin
            if (rst == 1'b1 || $past(Q) == MAX_COUNT)
              assert (Q == 8'b0);
            else if ($past(rst) == 1'b0)
              assert (Q - $past(Q) == 1);
          end
        end
```

```
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
- Propiedades multi-ciclo
- 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 produndidad 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  Trying induction in step 198..
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  Trying induction in step 18..
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

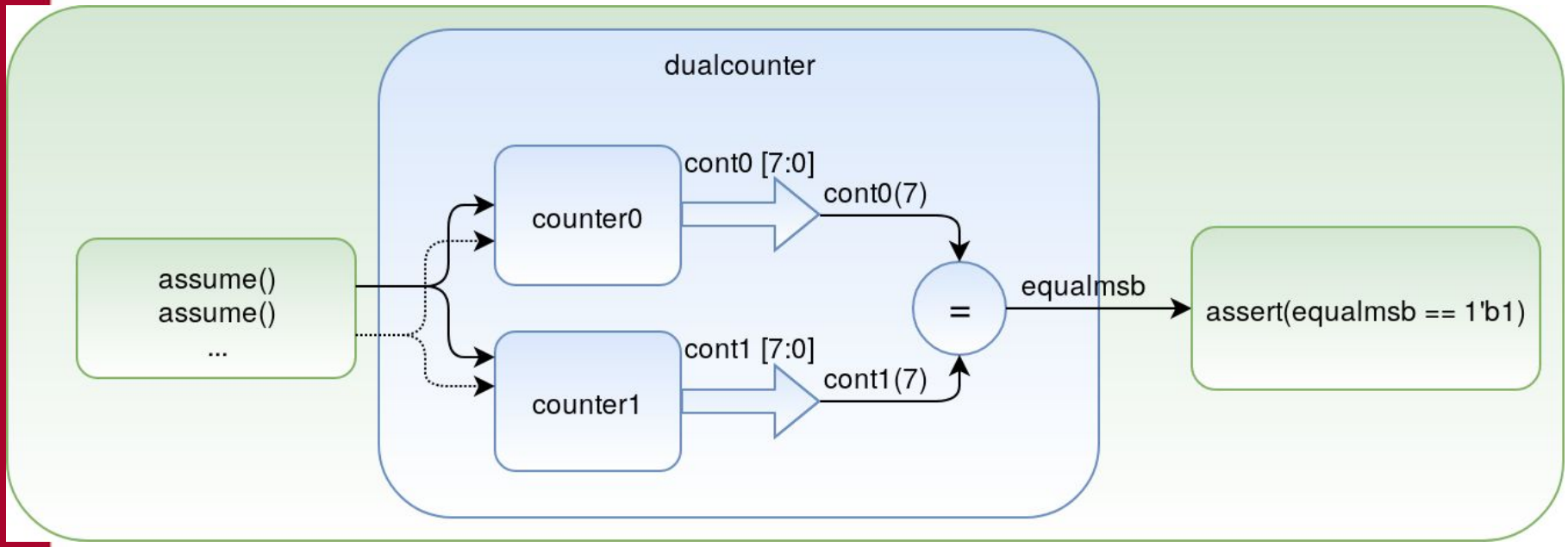
```
[options]
mode prove
depth 20
```

```
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 count0: 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 (Verilog)

```
`ifdef FORMAL
  reg f_past_valid;

  initial
    f_past_valid <= 1'b0;

  always @(posedge clk)
    f_past_valid <= 1'b1;

// Force a reset in the first cycle
  always @(*)
    if (f_past_valid == 1'b0)
      assume(rst == 1'b1);

// Output 'equalmsb' should always
// be '1'
  always @(*)
    assert (equalmsb == 1'b1);
`endif
```

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.copy:
dualcounter_properties.sv:31
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
```

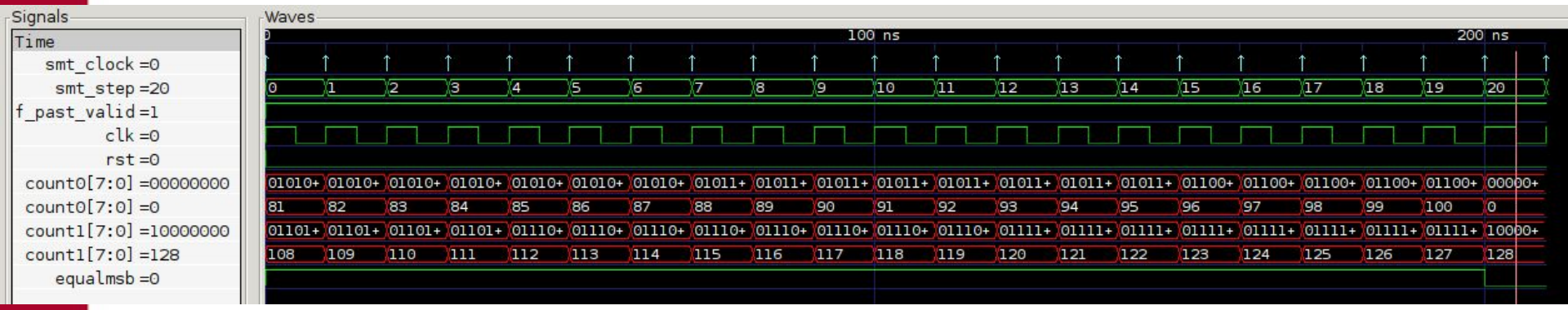
k-induction

Ejemplo 3

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

Propiedades (Verilog)

```
// Output 'equalmsb' should always
// be '1'
  always @(*)
    assert (equalmsb == 1'b1);
`endif
```



Dos posibles soluciones

a)

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

basecase: PASS

induction: PASS

b)

Propiedades (Verilog)

```
// Output 'equalmsb' should always
// be '1'. count0 and count1 should
// always match
always @(*)
  begin
    assert (equalmsb == 1'b1);
    assert (count0 == count1);
  end
```

basecase: PASS

induction: PASS

Dos posibles soluciones

a)

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

basecase: PASS

induction: PASS

a???)

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

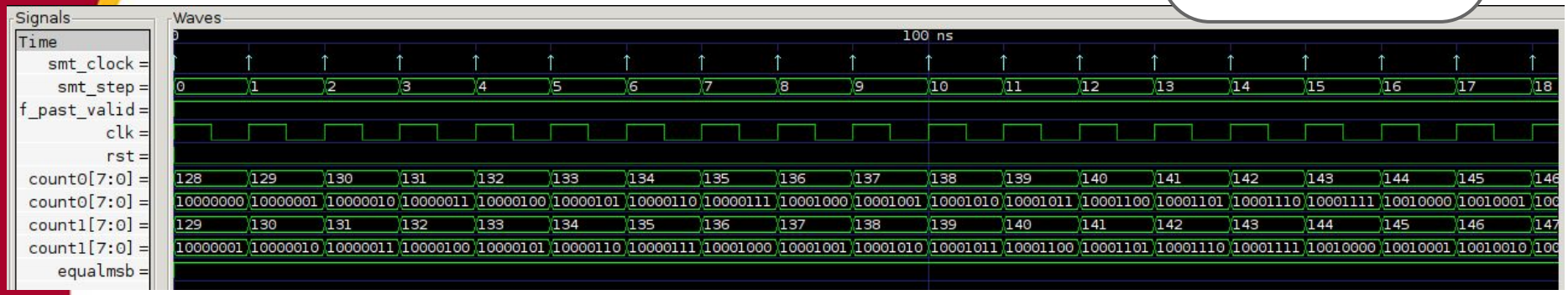
basecase: PASS

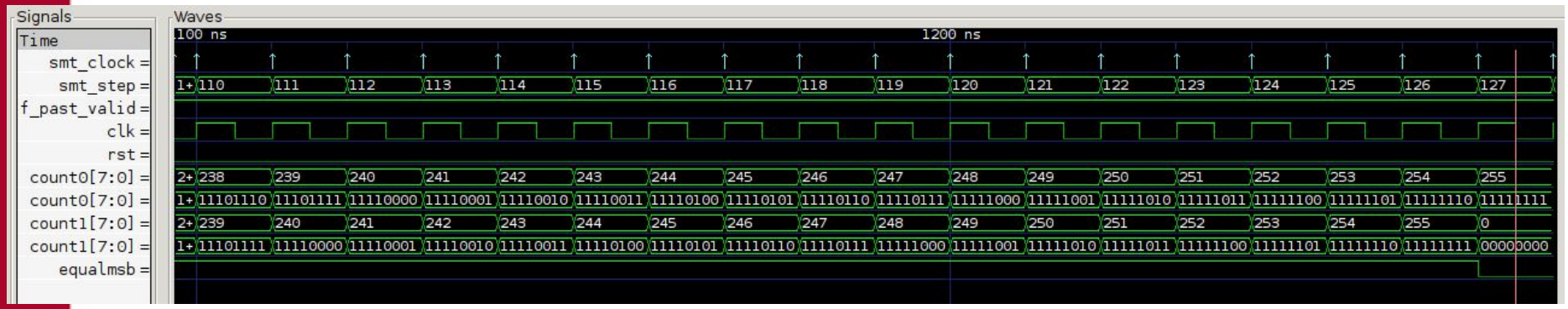
induction: FAIL

k-induction

Ejemplo 3

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

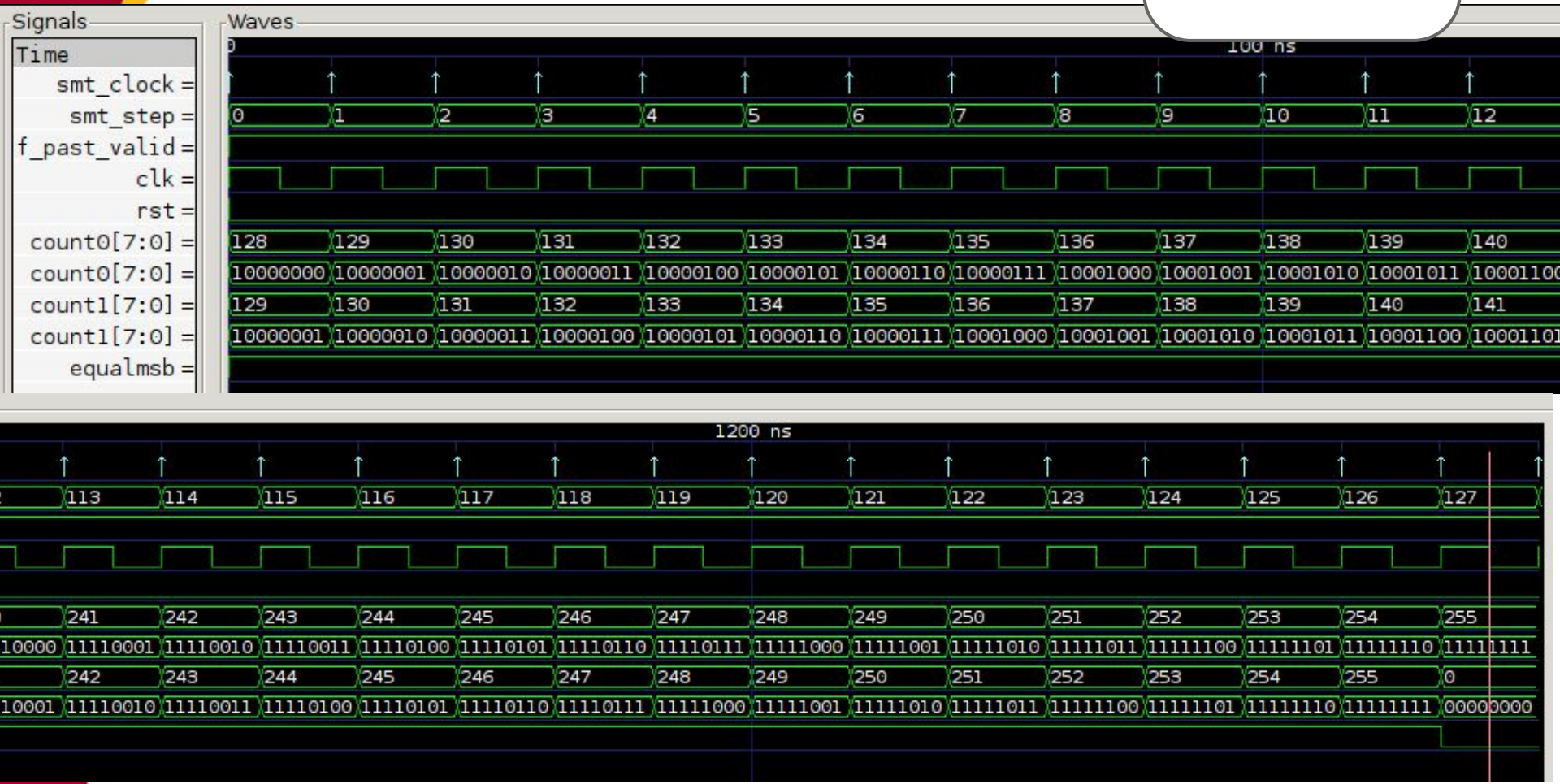




k-induction

Ejemplo 3

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



Contenido

- Motivación
- Definición
- Propiedades y suposiciones
- Propiedades multi-ciclo
- 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
```

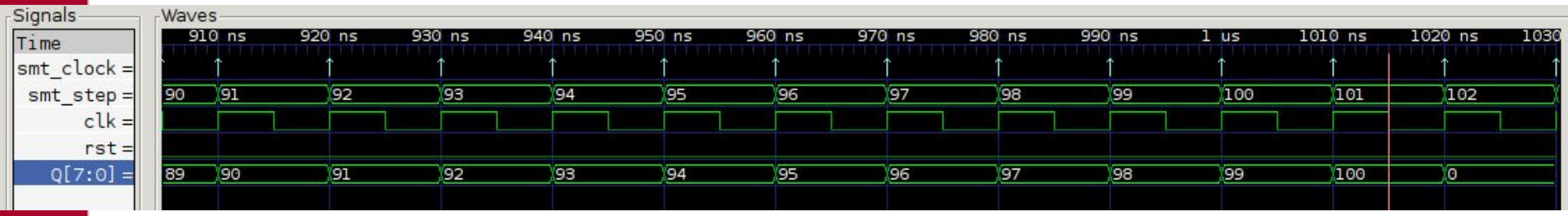
```
always @(posedge clk)
begin
  cover(Q == MAX_COUNT);
end
```

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
- Propiedades multi-ciclo
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

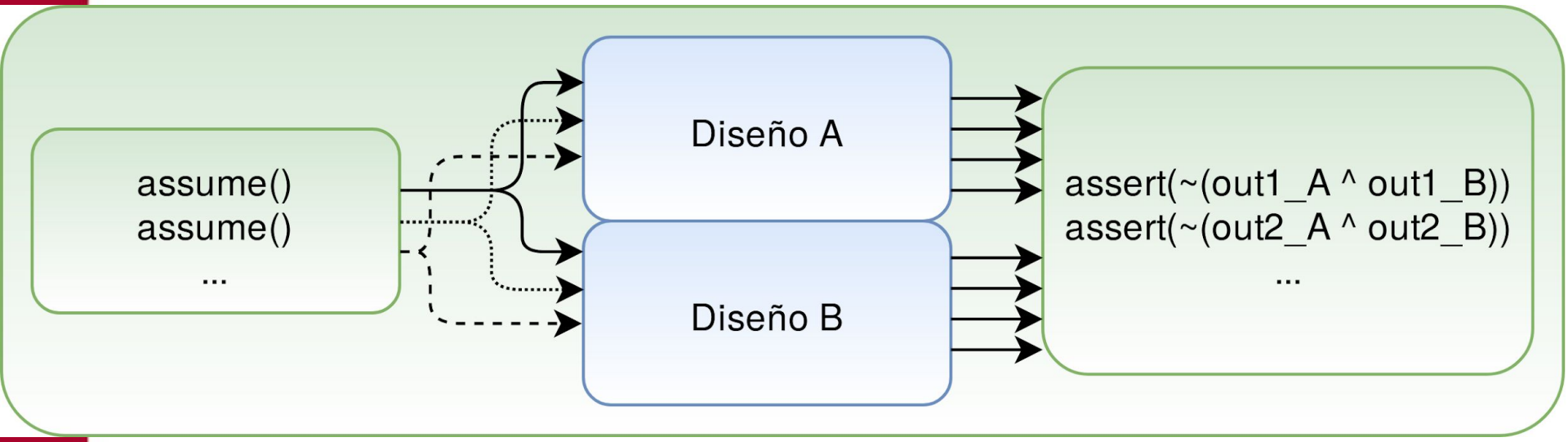
Comprobar que dos diseños se comportan igual

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
- Propiedades multi-ciclo
- 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 reg/wire que añadamos a las propiedades (f_past_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
- Propiedades multi-ciclo
- Bounded Model Checking
- k-induction
- Cover
- Equivalence Checking
- Conclusiones
- Bibliografía

Bibliografía

- Gisselquist, Dan, An Introduction to Formal Methods.
- 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 y suposiciones de un circuito?
- ¿Para qué sirve `$past` 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?