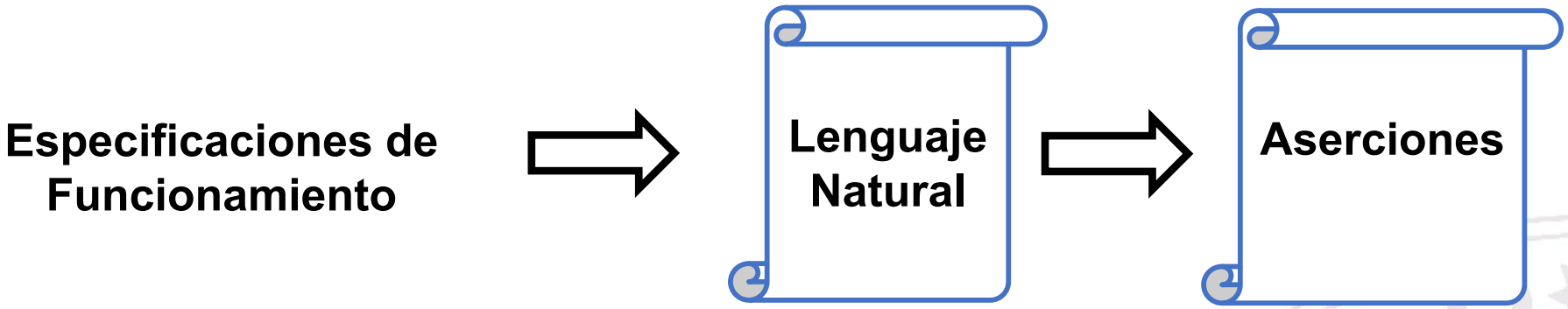
Diseño Digital I

Introducción a los lenguajes de aserciones. PSL

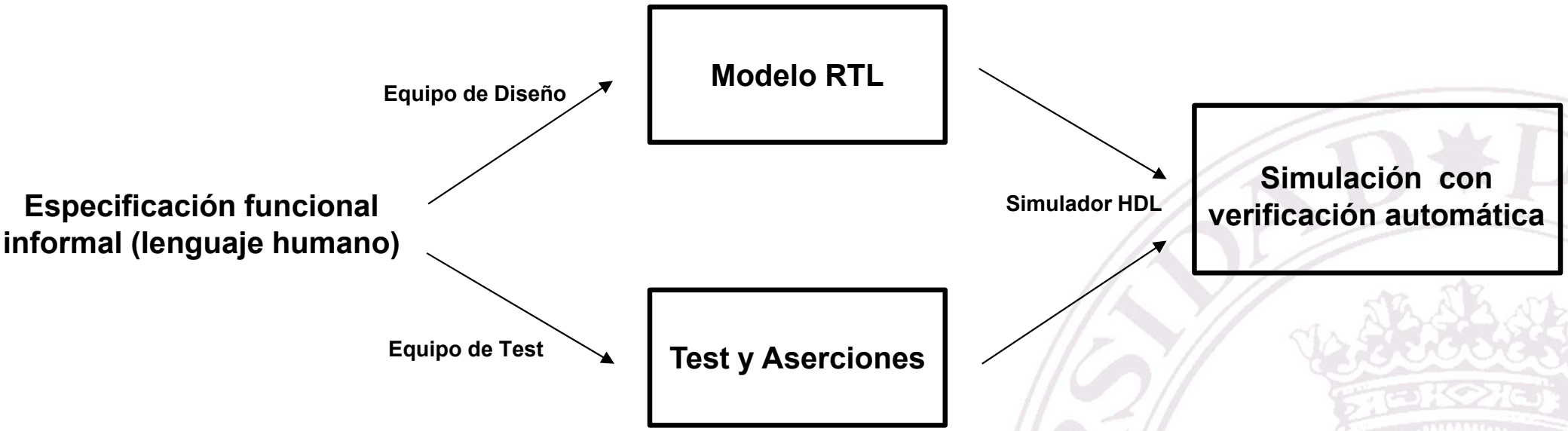
Lenguajes de aserciones



Ventajas:

- Resolución de ambigüedades
- Assertion Based Verification (ABV)
- Verificación formal

Metodología de test con lenguajes de aserciones



PSL

PSL, *Property Specification Language*, es un **Lenguaje de Aserciones**

- Estándar de IEEE
- **PSL** emplea la sintaxis del lenguaje subyacente (VHDL, System Verilog, System C)
- Uso dentro de modelos VHDL (RTL y tests):
  - con meta-comentarios (-- psl)
  - directamente, a partir de VHDL-2008 (no soportado por *Questa Sim*)
  - encapsulado en unidades de código específicas (*vunits*)
- El lenguaje de aserciones más utilizado es parte del ecosistema ***SystemVerilog*** y se denomina **SVA** (*SystemVerilog Assertions)*
  - **SVA** es más complejo y potente que **PSL**, pero su uso requiere ciertos conocimientos *avanzados* de *SystemVerilog*

Especificación funcional con PSL

PSL permite formular **secuencias** y **propiedades** para modelar los requisitos funcionales de los sistemas digitales

Secuencias:

- Una secuencia PSL define el valor que toman un conjunto de señales durante una serie de instantes de tiempo

Propiedades:

- Las propiedades se definen normalmente como relaciones entre dos o más secuencias
- La *relación de implicación* es la más comúnmente empleada para definir propiedades que representan funcionalidad

```
{a} |=> {b;c}

always {a} |=> {b;c}
```

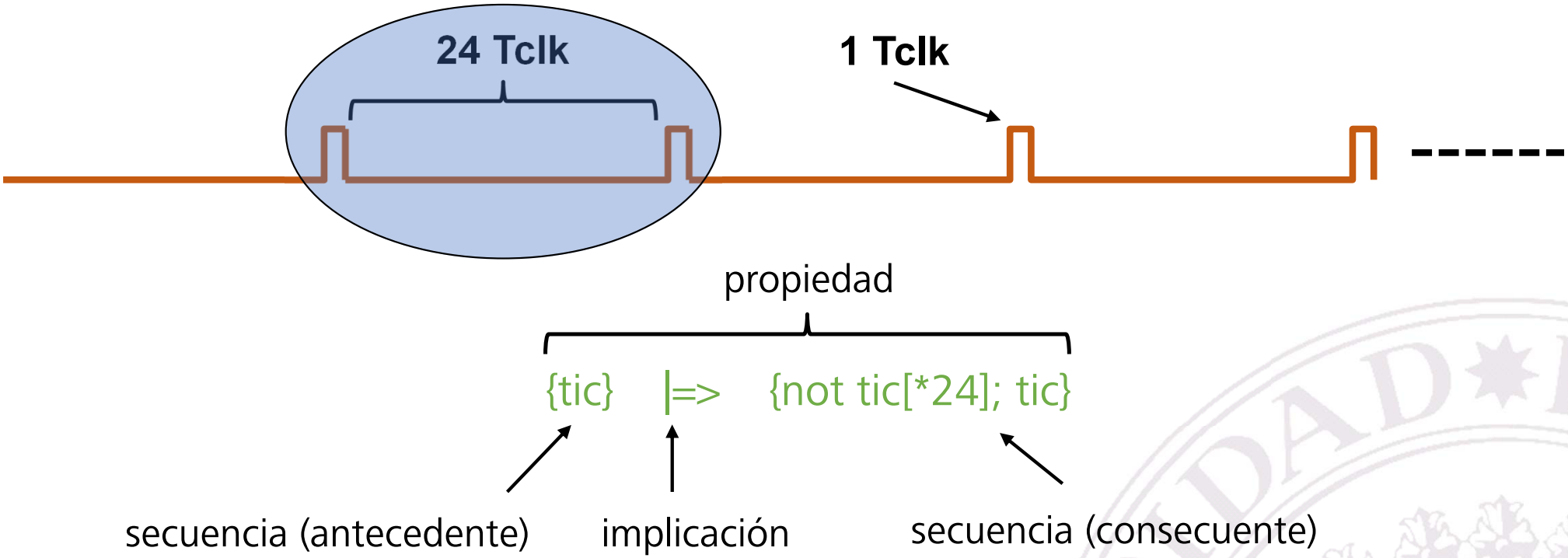
Aserciones:

- La aserción es un comando PSL (assert) que se aplica sobre una propiedad
  - Ordena a un simulador HDL que compruebe el cumplimiento de la propiedad durante la ejecución de una simulación

```
assert always {a} |=> {b;c};
```

  - Además del comando *assert*, en PSL hay otros, menos importantes, que sirven a otros propósitos

Ejemplo: timer (I)



- **Secuencia:** descripción del valor de una o más señales durante una serie, finita o indeterminada, de instantes de tiempo (periodos de reloj) de un sistema digital síncrono
- **Propiedad:** secuencia o relación entre dos o más secuencias de señales

- **La funcionalidad** de un sistema digital se puede expresar como un conjunto de **propiedades**

Ejemplo: timer (II)


```
always {tic} |=> {tic}    |=>    {not tic[*24]; tic} @ rose(clk)
                                                    @ rising_edge(clk)
                                                    @ clk'event and clk = '1'
```

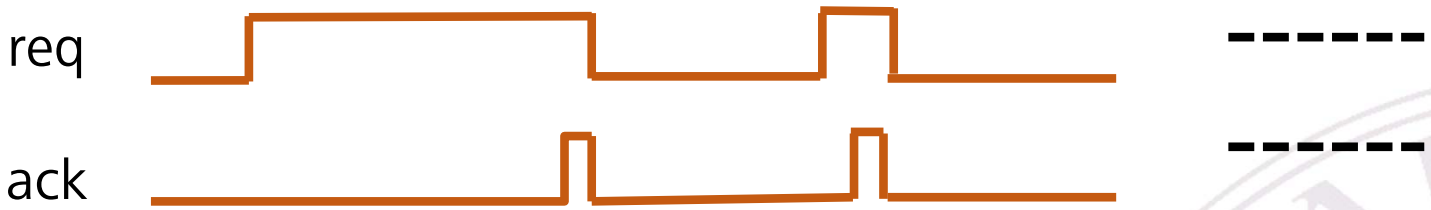
Ejemplo: timer (III)


```
assert always {tic} |=> {not tic[*24]; tic} @ rose(clk)
report "el tic no se genera correctamente";

assert {not tic[*0:99]} |=> {tic} @ rose(clk)
report "no se genera el tic";
```

Ejemplo: rq/ack (LTL)

Siempre que se active req, debe permanecer activa hasta la activación de ack. La duración de ack debe ser de un período de reloj del sistema y al desactivarse debe hacerlo también req, de manera simultánea.




```
assert always req ->  req until_      @ rose(clk)
                      ack
report "no ack después de req";

assert always ack -> next not ack @ rose(clk)
report "ack no dura un período de reloj";
```

Especificación funcional con PSL (I)

Las secuencias requieren la definición del evento que determina el transcurso del tiempo en el sistema, normalmente un flanco de la señal de reloj

```
assert always req ->  req until_ ack @rose(clk)
report "no ack después de req";
```

```
default clock is  "clk'event and clk = '1'"        -- o rose(clk), o rising_edge(clk)
.....
assert always req ->  req until_ ack ~~@rose(clk)~~
report "no ack después de req";
```

Ejemplo de test con lenguajes de aserciones (I)

Diseño y verificación de un timer que genera un *tic* cada 25 periodos de la señal de reloj (trabaje sobre la carpeta ***timer***, cree un proyecto y añada los ficheros *timer_assert.vhd* y *test_timer_assert.vhd*):

1.- Abra el fichero *timer_assert.vhd* con el editor de Questa
Contiene el modelo de un timer parametrizable

2.- Analice, primero, el modelo rtl del timer –un contador parametrizable de módulo 25 con salida de fin de cuenta- y deténgase después en el código PSL introducido mediante *meta-comentarios*

```
entity timer_25 is
generic(div_param: natural := 25);   -- periodo de tic s -> 5 ms

port(nRst:      in  std_logic;
     clk:       in  std_logic;
     tic_25:    out std_logic);

end entity;

architecture rtl of timer_25 is
  signal cnt: std_logic_vector(4 downto 0);

  -- CODIGO___PSL con metacomentarios
  -- psl default clock is (clk'event and clk = '1');
  -- psl sequence tic_N(numeric N) is {(not tic_25)[*(N-1)];tic_25};
  -- psl property tic_N_periodico(numeric N) is (always {tic_25} |=> tic_N(N));
  -- psl assert tic_N_periodico(div_param);
```

a.- Los comentarios que comienzan con -- psl son para Questa código PSL empotrado en código VHDL

b.- Observe que hay elementos sintácticos VHDL –porque PSL emplea una sintaxis propia genérica y aprovecha operadores y elementos del HDL donde se empotra

Ejemplo de test con lenguajes de aserciones (II)

```
entity timer_25 is
generic(div_param: natural := 25);    -- periodo de tic s -> 5 ms

port(nRst:      in  std_logic;
     clk:       in  std_logic;
     tic_25:    out std_logic);

end entity;

architecture rtl of timer_25 is
  signal cnt: std_logic_vector(4 downto 0);

  -- CODIGO___PSL con metacomentarios
  -- psl default clock is (clk'event and clk = '1');
  -- psl sequence tic_N(numeric N) is {(not tic_25)[*(N-1)];tic_25};
  -- psl property tic_N_periodico(numeric N) is (always {tic_25} |=> tic_N(N));
  -- psl assert tic_N_periodico(div_param);
```

c.- Observe que se declara un *default clock* –porque PSL evalúa los niveles de las señales que componen las secuencias síncronamente y resulta obligado identificar cuál es la señal de tiempo de las aserciones

-Para aclarar el significado del carácter síncrono de PSL: Puesto que en el ejemplo el *default clock* es el flanco de subida de *clk*, la fórmula de la secuencia {(not tic_25)[*24]; tic_25} expresa que *tic_25* está a nivel bajo en 24 flancos de subida consecutivos de *clk* y a nivel alto en el siguiente flanco de subida

d.- Observe que PSL dispone de elementos (***sequence*** o ***property***) que permiten estructurar el código que define una propiedad –esto resulta extremadamente útil para definir propiedades complejas y, también, para crear secuencias y propiedades parametrizables

Ejemplo de test con lenguajes de aserciones (III)

```
sequence tic_N(numeric N) is {(not tic_25)[*(N-1)];tic_25};

property tic_N_periodico(numeric N) is (always {tic_25} |=> tic_N(N));

assert tic_N_periodico(div_param);
```

e.- Estructura más frecuentemente utilizada para modelar funcionalidad:
secuencia precedente *tic_25* que implica |=> una secuencia consecuente *tic_N(N)*

-Expresa la relación causa-efecto y su ocurrencia secuencial en el tiempo:

"tras la activación de una entrada sucede una determinada –y conocida- secuencia de señales durante un determinado número de instantes de tiempo"

-El operador de implicación indica que, para que la propiedad se cumpla, a la ocurrencia efectiva de la causa debe seguir necesariamente la ocurrencia precisa del efecto, pero –**y esto es muy importante**-, también que si no se da la causa, la propiedad se cumple **siempre** (ejecución *vacua*)

-El operador *always* indica que la vigencia de la propiedad es continua, esto es, que debe observarse y detectarse la ocurrencia de la propiedad en **todos** los instantes de tiempo.

{tic_25} |=> tic_N(N) significa que tic_N(N) solo debe necesariamente suceder si **tic_25 = '1' en el primer ciclo de reloj**

f.- La sentencia *assert* instruye al simulador VHDL para que vigile el cumplimiento de la propiedad

Ejemplo de test con lenguajes de aserciones (IV)

Ejecución de una simulación con aserciones

1.- Compile los ficheros *timer_assert.vhd* y *test_timer_assert.vhd*

2.- Ordene el arranque de una simulación (solo el **arranque**, no la ejecución)

3.- Active la venta de aserciones (View > Coverage > Assertions) y habilite su ejecución si no lo está ya (desde la ventana Assertions, botón dcho., Enable)

4.- Emplee el fichero *timer_assert.do* para configurar el visor de formas de onda

5.- Ejecute una simulación y revise los resultados -que serán explicados por el profesor

Fallo de una aserción

1.- Descomente el código del proceso que maneja el discurso del test en *test_timer_assert.vhd* –no le costará identificarlo

2.- En el modelo del *timer*, modifique la aserción para añadir un mensaje de aviso en caso de fallo de la propiedad:

```
-- psl assert tic_N_periodico(div_param)
--      report "error en el periodo de tic_25";
```

3.- Observe que la línea que añade no lleva la etiqueta *psl* porque el código es una extensión de la sentencia *assert*

4.- Recompile los ficheros y re-arranque la simulación

5.- Ejecute la simulación y revise los resultados -que serán explicados por el profesor

Ejemplo de test con lenguajes de aserciones (V)

Efecto del reset en las aserciones

Si se desea evitar el fallo de una aserción porque en el transcurso de la comprobación de una propiedad se activa el *reset* del circuito, hay que añadir una cláusula de anulación a la aserción

1.- En el modelo del *timer*, modifique la aserción para añadir una anulación de la propiedad en caso de activación del reset:

```
property tic_N_periodico(numeric N) is always ({tic_25} |=> tic_N(N)) abort fell(nRst);
```

Detalles:

- La cláusula *abort* –puede ponerse también *async_abort*- indica que debe suspenderse la vigilancia de la propiedad si *fell(nRst)*
- *fell* es una función predefinida de PSL que, en el contexto en que se utiliza aquí, se verifica cuando la señal que se le pasa como parámetro (*nRst*) pasa de nivel alto a nivel bajo asíncronamente –en otros contextos hace referencia a flancos de bajada síncronos

MUY IMPORTANTE: La cláusula *abort afecta únicamente a la sub-propiedad* {tic_25} |=> tic_N(N)}. Si se aplica a la propiedad completa, (always {tic_25} |=> tic_N(N)) el efecto equivale a la anulación definitiva de la aserción

2.- Recompile los ficheros y re-arranque la simulación

3.- Ejecute la simulación y revise los resultados -que serán explicados por el profesor

Si lo desea, pruebe el efecto de anular la ejecución de la aserción completa

Ejemplo de test con lenguajes de aserciones (VI)

Ejecución vacua de aserciones

En este último apartado del ejercicio se va a probar un cambio grosero que avería el modelo del timer para aprender más cosas sobre las aserciones:

1.- En el modelo del timer, modifique la sentencia que activa tic_25

```
tic_25 <= '0';
```

2.- Recompile los ficheros y re-arranque la simulación

3.- Ejecute la simulación y revise los resultados

Observe que:

- La aserción NO FALLA porque en todos los instantes de tiempo el antecedente no se cumple –tic_25 nunca vale '1'- y, por tanto, solo se producen EJECUCIONES VACUAS de la aserción: la implicación es TRUE siempre que el antecedente es FALSE

- Para detectar estos casos se pueden emplear sentencias COVER (de cobertura) cuyo propósito es contar el número de ocurrencias de una determinada secuencia durante una simulación

4.- Añada al modelo del *timer* la siguiente sentencia PSL: -- psl cover {tic_25};

5.- Deshaga el cambio en la sentencia de asignación de tic_25

6.- Recompile los ficheros y re-arranque la simulación. Active: View > Coverage > Cover Directives y desde esa ventana, boton derecho, Display Options > Recursive Mode

7.- Ejecute la simulación y revise los resultados que serán comentados por el profesor

Revisión de PSL. Estilos LTL y SERE

Dos estilos sintácticos para la definición de secuencias y propiedades: LTL y SERE

- El estilo LTL se asemeja más al lenguaje natural que el SERE, pero hay secuencias que no pueden construirse en LTL

```
assert always req ->  req until_ ack @ rose(clk)
report "no ack después de req";
```

- El estilo SERE permite describir cualquier secuencia de señales, pero las expresiones suelen más extensas y complejas que su equivalente en LTL –si éste existe

```
assert always {req} |=>  {req[*]; ack and req} @ rose(clk)
report "no ack después de req";
```

Revisión de PSL. Estilo LTL (I)

- Nombre derivado del lenguaje LTL (*Linear Temporal Logic*) en que se basa

- Operadores:
  - *always*, *never*, *eventually!*
  - *until*, *before*
  - *next*, *next_a*, *next_e*
  - *next_event*, *next_event_a*, *next_event_e*
  - *->* (*implicación*)
  - *Existen versiones weak y strong de muchos de estos operadores*

- *Las propiedades formuladas en estilo LTL se componen con los operadores anteriores, señales, expresiones lógicas y funciones predefinidas PSL*
  - *Funciones predefinidas: prev, stable, rose, fell… entre otras*

- *Hay algunas secuencias (pocas) que no pueden expresarse en estilo LTL*

Revisión de PSL. Estilo LTL (II)

Operador	Ejemplo	Descripción
implicación lógica ->	a and not b -> c	Si a = '1' y b = '0', entonces, en el mismo ciclo de reloj, c ='1' (el consecuente solo tiene que verificarse en el primer ciclo en el que se verifique el antecedente). LHS tiene que ser una expresión booleana
always	always a and not b -> c	Si a = '1' y b = '0', entonces, en cada ciclo de reloj en el que ocurre, c ='1'
never	never a and not b	Nunca a = '1' y b = '0' en el mismo ciclo de reloj
never	never a and not b -> c	always not(a and not b -> c) always a and not b -> not c
next	a -> next b	Si a se activa, entonces, en el siguiente ciclo debe activarse b
next[n]	a -> next[n] b	Si a se activa, entonces, n ciclos después debe activarse b
next_a[n:m]	a -> next_a[n:m] b	Si a se activa, entonces, b se activa n ciclos después y permanece activa al menos hasta m ciclos después (a=all)
next_e[n:m]	a -> next_e[n:m] b	Si a se activa, entonces, b se activa al menos una vez entre n y m ciclos después (e=exist)
next_event(b)(c)	a -> next_event(b)(c)	Si a se activa, entonces, c debe activarse en el primer ciclo en el que se active b
next_event_a(b)[n:m](c)	a -> next_event_a(b)[n:m](c)	Si a se activa, entonces, c debe activarse en todos los ciclos en los que se active b entre el n-simo y el m-simo
next_event_e(b)[n:m](c)	a -> next_event_e(b)[n:m](c)	Si a se activa, entonces, c debe activarse en al menos uno de los ciclos en los que se active b entre el n-simo y el m-simo

Revisión de PSL. Estilo LTL (III)

Operador	Ejemplo	Descripción
until	a -> next b until c	Si a se activa, en el siguiente ciclo se activa b hasta que se active c
until_	a -> next b until_ c	Si a se activa, en el siguiente ciclo se activa b hasta que se active c, incluyendo el ciclo en el que se activa c
before	a -> b before c	Si a se activa, b debe activarse al menos una vez antes de que se active c
before_	a -> b before_ c	Si a se activa, b debe activarse al menos una vez antes de que se active c, pudiendo hacerlo incluso en el ciclo en el que se activa c
until!, until!_, before!, before!_, next!, next!_, next_event!, next_event_a!, next_event_e!	a -> next b before! c	Versiones *strong* de los operadores *weak*
eventually!	a -> eventually! b	Si a se activa, entonces b debe activarse en algún momento

Revisión de PSL. Estilo LTL (IV)

Funciones predefinidas	Ejemplo	Descripción
rose(), fell()	rose(a) -> next b until fell(c)	Si hay un flanco de subida en a, entonces, b se activa hasta que haya un flanco de bajada en c
stable(b)	a -> next stable(b)	Si a se activa, en el siguiente ciclo b tiene que tener el mismo valor que en el anterior (en el que se ha activado a)
prev(b, n)	a -> next[5] b = prev(c,2)	Si a se activa, 5 ciclos después b debe tomar el mismo valor que tomó c 2 ciclos antes

Ejercicio LTL (I)

La siguiente aserción vigila el efecto de la activación de la entrada *start d*el *master SPI :*

"Si se activa ***start***, a nivel alto, y la salida ***rdy*** está a nivel alto, comienza una transferencia (en el siguiente ciclo de reloj, ***nCS*** se activa a nivel bajo)"

```
ass_start_then_nCS_low: assert always ((start and rdy) -> next(not nCS))      -- ASSERT basica LTL
                        report "No se atiende la orden de start" ;
```

Ejercicio LTL (II)

Inclusión de código PSL en VHDL con meta-comentarios:

-- *psl* *(código psl)*

*A*ñadiendo definición de reloj de muestreo:

```
-- psl default clock is (clk'event and clk = '1');
-- psl ass_start_then_nCS_low: assert always ((start and rdy) -> next(not nCS))  -- ASSERT basica LTL
--                                          report "No se atiende la orden de start" ;
```

Descripción del funcionamiento esperado:

-- Si en el instante siguiente al de activación de ***start*** con ***rdy*** a nivel alto, ***nCS*** se pone a nivel bajo, la aserción PASA; si no, la aserción FALLA y se escribe en la consola el mensaje de error

-- Si no se activa ***start*** o se activa con ***rdy*** a nivel bajo, la aserción PASA; esto se denomina ejecución ***vacua*** de la aserción

Ejercicio LTL (III)

Ejercicio Questa:

NOTA: Alguna de las siguientes operaciones se hará bajo la guía, o con la ayuda, del profesor

*1.- Cree en Questa el proyecto Ejercicios_PSL (trabaje sobre la carpeta **SPI**):*

2.- Revise el código del fichero del primer ejercicio: gen_trazas_start_then_nCS_low.vhd

- Fíjese en el Código PSL incluido mediante meta-comentarios

- El Código en el cuerpo de arquitectura genera trazas de prueba para ejercitar la aserción y comprobar si funciona como se espera. Revíselo y haga un dibujo de las trazas

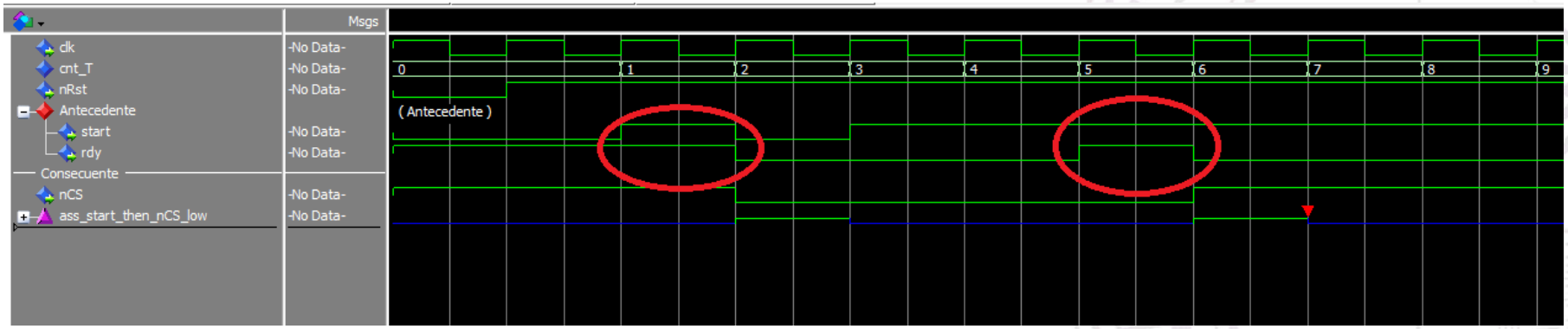
3.- Compile el fichero y arranque la simulación (como si fuera un test-bench)

4.- Active la ventana de aserciones: View -> Coverage -> assertions

5.- Active las aserciones en el menú assertions y Display options > Recursive mode

6.- Cargue en la ventana Waves el fichero gen_trazas_start_then_nCS_low.do

7.- Ejecute una simulación; el profesor le comentará los resultados de la misma



Ejercicio LTL (IV)

- Para saber cuantas veces se ha dado la comprobación de una condición con una sentencia assert durante una simulación puede utilizarse la orden *cover*.

- Sirve también para diferenciar el caso en que se han dado las situaciones que vigila una sentencia *assert* con éxito de los casos en los que únicamente ha habido ejecuciones vacuas.

En nuestro ejemplo:

1.- Añada al fichero gen_trazas_start_then_nCS_low.vhd el siguiente código:

-- psl cover_start_rdy: cover {(start and rdy)};

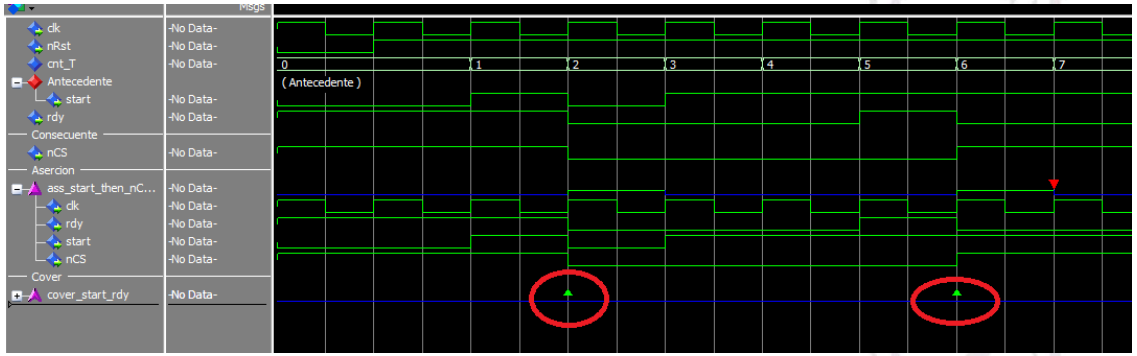
2.- Compile el fichero y arranque la simulación (como si fuera un test-bench)

3.- Active la ventana de directivas de cobertura: View -> Coverage -> Cover Directives

4.- Active el análisis de cobertura en el menú Cover Directives: Configure Directives > Counting(enable)

5.- Cargue en la ventana Waves el fichero gen_trazas_start_then_nCS_low_with_cover.do

6.- Ejecute una simulación; el profesor le comentará los resultados de la misma



Ejercicio LTL (V)

- La sentencia *assume* sirve para especificar propiedades que se van a cumplir durante el funcionamiento de un circuito (muy útil para las herramientas de verificación formal)

- La sentencia *asume_guarantee* especifica una propiedad que se va a cumplir, pero se pide que se compruebe como una aserción.

Por ejemplo: "Los pulsos de ***start*** duran un periodo de reloj" se formularía de la siguiente manera:

```
asm_gu_start_tic: assume_guarantee always (start -> next(not start))
                                report "NOTE: Pulso de start de más de 1 Tclk" ;
```

En nuestro ejemplo:

1.- Añada al fichero gen_trazas_start_then_nCS_low.vhd el código anterior
2.- Compile el fichero y arranque la simulación (como si fuera un test-bench)
3.- Cargue en Waves el fichero gen_trazas_start_then_nCS_low_with_cover_with_ass_gu.do
4.- Ejecute una simulación y revise los resultados

Revisión de PSL. Estilo SERE (I)

- SERE: *Sequential Extended Regular Expressions*

- Operadores
  - *De repetición:* [*], [+], [=], [->]
  - *De concatenación:* ':' *y* ';'
  - *De composición de secuencias:* within, &&, & *y* |
  - |-> *y* |=> *(implicación)*
  - *Existen versiones* weak *y* strong (!) *de secuencias SERE*

- *Las propiedades formuladas en estilo SERE se componen con los operadores anteriores, señales, expresiones lógicas y funciones predefinidas PSL (prev, stable, rose, fell… entre otras)*

Revisión de PSL. Estilo SERE (II)

Operador	Ejemplo	Descripción
[*]	{a[*]}	Activación de a durante un número indeterminado de ciclos consecutivos (pueden ser 0)
[+]	{a[+] }	Activación de a durante uno o más ciclos consecutivos
[*n]	{a[*n] }	Activación de a durante n ciclos consecutivos
[*n:m]	{a[*n:m] }	Activación de a durante entre n y m ciclos consecutivos
[*n:inf]	{a[*n:inf] }	Activación de a durante al menos n ciclos consecutivos
[=n]	{a[=n] }	Activación de a durante n ciclos no necesariamente consecutivos. Solo aplica a booleanos.
[->n]	{a[->n] }	Activación de a durante n ciclos no necesariamente consecutivos, pero el último coincide con la finalización de la secuencia (go to n) . Solo aplica a booleanos.
[->]	{a[->] }	Es lo mismo que {a[->1] } (go to al siguiente ciclo en el que se active a)
[=n:m]	{a[=n:m] }	Activación de a durante entre n y m ciclos no necesariamente consecutivos
[->n:m]	{a[->n:m] }	Activación de a durante entre n y m ciclos no necesariamente consecutivos, pero el último coincide con la finalización de la secuencia
;	{a[*5]; b}	Activación de a durante 5 ciclos consecutivos, en el sexto se activa b (concatenación)
:	{a[*5]: b[*3]}	Activación de a durante 5 ciclos consecutivos, b se activa del quinto al séptimo (concatenación con solapamiento)

Revisión de PSL. Estilo SERE (III)

Operador	Ejemplo	Descripción
|	{{a[=5]} | {b[*]}}	Activación de a en 5 ciclos no necesariamente consecutivos o activación de b un número indeterminado de ciclos (o ambas, pero la secuencia termina al acabar cualquiera de ellas)
&&	{{a[=5]} && {b[*]}}	Activación de a en 5 ciclos no necesariamente consecutivos durante la activación de b, terminando ambas sub-secuencias en el mismo ciclo
&	{{a[=5]} & {b[*]}}	Activación de a en 5 ciclos no necesariamente consecutivos . Activación de b un número indeterminado de ciclos, cada sub-secuencia es independiente
within	{{a[=5]} within {b[*]}}	Activación de a en 5 ciclos no necesariamente consecutivos durante la activación de b. Equivale a {[*];a[=5];[*]} && {b[*]}
|=>	{a; b[*3]} |=> {c}	Si se activa a (en el primer ciclo) y luego b tres ciclos consecutivos, entonces, en el quinto ciclo se activa c (implicación SERE)
|->	{a; b[*3]} |-> {c}	Si se activa a (en el primer ciclo) y luego b tres ciclos consecutivos, entonces, en el cuarto ciclo se activa c (implicación SERE con solapamiento)
always	always {a; b[*3]} |=> {c}	Siempre que se active a y luego b durante tres ciclos consecutivos, entonces, en el siguiente ciclo, se activa c
(skip)	always {a; [+];b[*3]} |=> {c;[*n:m];d}	Siempre que se active a, y al menos un ciclo después se active b 3 veces consecutivas, entonces, en el siguiente ciclo se activa c y, tras entre n y m ciclos, se activa d

Revisión de PSL. Estilo SERE (IV)

Funciones predefinidas	Ejemplo	Descripción
rose(), fell(), stable(), prev()	--	Las mismas que se definieron el LTL
ended()	ended({a; b[*3]})	Devuelve true en el ciclo en el que se cumpla la secuencia {a; b[*3]}
ended()	always {a; b[*3]} |=> {c} always {c}|-> prev(ended({{a; b[*3]} }))	Siempre que se active a y luego b durante tres ciclos consecutivos, entonces, en el siguiente ciclo, se activa c Siempre que se active c es porque en el ciclo anterior se ha cumplido la secuencia {a; b[*3]}

Revisión de PSL. Mezcla de estilos LTL y SERE

Operador	Ejemplo	Descripción
|-> o |=>	{a; b[*3]} |=> c before d	LHS tiene que ser SERE. RHS puede ser cualquier propiedad
->	a -> {[*]; c; [*]; d};	LHS tiene que ser expresión booleana. RHS puede ser SERE
eventually!	{a} |-> eventually! {c; [*]; d};	El operando de eventually! puede ser SERE
until, until!	{a} |-> {[*]; c[*]} until d	LHS puede ser SERE. RHS tiene que ser expresión booleana
next	{a} |=> next {[*]; c[*]; d}	Después de next puede haber un SERE
next_event_e, next_event_a	{a} |-> next_event_e(b)[1:3]({[*]; c[*]; d})	El primer operando, b, tiene que ser expresión booleana. El segundo puede ser SERE
|-> , |=>, ->	a -> eventually! {c; [*]; d} {a} |=> {[*]; c[*]} until d {a} |-> next {[*]; c[*]; d}	En los 4 ejemplos anteriores pueden utilizarse cualquiera de los operadores de implicación

Ejercicio SERE (I)

La siguiente aserción vigila que durante las transferencias de lectura en el master SPI se generen 2 pulsos de *ena_rd.*

"Si se activa ***start***, a nivel alto, y la entrada ***nWR_RD*** y la salida ***rdy*** están a nivel alto, comienza una transferencia (que dura mientras ***nCS*** está a nivel bajo) en la que se producen 2 pulsos de ***ena_rd*** de duración un período de reloj"

Fórmula SERE:

```
ass_2_tics: assert always({start and rdy and nWR_RD} |=>
                          {{ena_rd}[->2] && {(not nCS)[*]};(not ena_rd)[*]:rose(nCS)} ) abort fell(nRst)
        report "No se han producido 2 pulsos de ena_rd en la lectura";
```

Ejercicio SERE (II)

```
ass_2_tics_ena_rd: assert always({start and rdy and nWR_RD} |=>
                                 {{ena_rd[->2]} && {(not nCS)[*]};(not ena_rd)[*]:rose(nCS)} ) abort fell(nRst)
                   report "No se han producido 2 pulsos de ena_rd en la lectura";
```

En el instante siguiente al de la orden de lectura

```
{start and rdy and nWR_RD} |=>  …
```

…deben producirse dos pulsos de **ena_rd** mientras **nCS** está a nivel bajo:

```
{ena_rd[->2]} && {(not nCS)[*]}
```

La longitud de esta secuencia viene determinada por la ocurrencia de 2 pulsos de **ena_rd** o porque **nCS** vuelva a 1 (en cuyo caso fallaría), después…

…**ena_rd** se mantiene inactiva hasta que se desactive **nCS**:

```
(not ena_rd)[*]:rose(nCS)
```

La comprobación del *consecuente* se anula automáticamente por la activación asíncrona del reset:

```
abort fell(nRst)
```

Ejercicio SERE (III)

Ejercicio Questa:

1.- Abra en Questa el proyecto Ejercicios_PSL

2.- Revise el código del fichero del segundo ejercicio: gen_trazas_ena_rd_1.vhd

- Observe que el Código PSL incluido mediante meta-comentarios corresponde a la aserción descrita

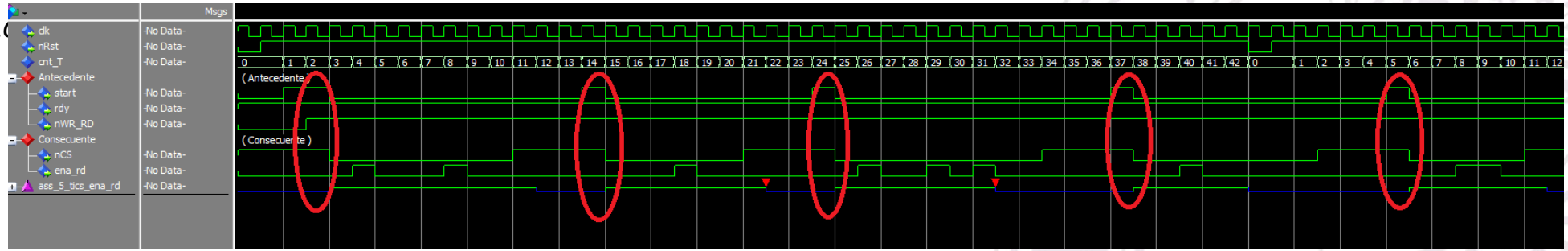
3.- Compile el fichero y arranque una simulación (como si fuera un test-bench)

4.- Emplee el fichero gen_trazas_ena_rd_1.do para configurar la ventana Waves

5.- Ejecute una simulación

6.- Revise los resultados de los cinco disparos de la aserción. Observe que solo hay una traza que cumpla las condiciones válidas, pero la aserción pasa 3 veces, en uno de los casos, erróneamente. Determine en cuál.

L



Ejercicio SERE (IV)

Ejercicio Questa:

1.- Revise el código de una nueva versión del fichero del segundo ejercicio: gen_trazas_ena_rd_2_a.vhd
- Observe que el Código PSL incluye la aserción errónea descrita anteriormente y una nueva

2.- Compile el fichero y arranque una simulación (como si fuera un test-bench)
3.- Emplee el fichero para configurar la ventana Waves
4.- Ejecute una simulación
5.- Revise la respuesta de las aserciones a los disparos
6.- Repita todo el proceso (del 1 al 5) con los ficheros gen_trazas_ena_rd_2_b.vhd y gen_trazas_ena_rd_2_b.do

Ejercicio SERE (V)

Ejercicio Questa:

1.- Revise el código de la última versión del fichero del segundo ejercicio: gen_trazas_ena_rd_3.vhd
- Observe que el Código PSL incluye las dos aserciones erróneas descritas anteriormente y una nueva

2.- Compile el fichero y arranque una simulación (como si fuera un test-bench)
3.- Emplee el fichero gen_trazas_ena_rd_3.do para configurar la ventana Waves
4.- Ejecute una simulación
5.- Revise la respuesta de las aserciones a los disparos
6.- Observe cómo la nueva aserción verifica correctamente la funcionalidad

El profesor le explicará los detalles que justifican el comportamiento de las aserciones y resumirá algunas conclusiones útiles relativas al caso

Vunit + Ejercicio vunit

PSL proporciona unidades de código, denominadas **vunit**, que permiten concentrar el conjunto de aserciones que completan la especificación funcional de un sistema o interfaz

Ejercicio Questa:

*1.- Cree el proyecto Ejer_Master_SPI en la carpeta **Master_SPI** e incluya en él los tres ficheros *.vhd (**master_spi_4_hilos**, **test_master_spi_4_hilos** y **agent_spi_sensor**).*

*2.- El código del fichero **check_master_spi_4_hilos.psl** contiene aserciones que especifican el funcionamiento del master SPI desarrollado en sesiones anteriores. La especificación corresponde, tanto al funcionamiento de la interfaz de control del módulo como al protocolo del bus SPI del acelerómetro. Revise el código (ábralo desde Questa aunque no esté incluido en el proyecto) y consulte con el profesor cualquier duda o curiosidad en relación con él.*

*3.- En Questa es necesario vincular el fichero que contiene la **vunit** PSL al modelo cuya funcionalidad describe, de la siguiente manera:*

- *Seleccione las propiedades del fichero master_spi_4_hilos.vhd y, en la pestaña VHDL, seleccione el fichero PSL check_master_spi_4_hilos.psl –en **Other VHL options***
- *Seleccione, si no lo está ya, la opción **Use vopt Flow** en la misma ventana*

*4.- Compile los tres ficheros y ejecute una simulación (utilice el fichero *.do adjunto para configurar Waves)*

El profesor le explicará los detalles que justifican el comportamiento de las aserciones y resumirá algunas conclusiones útiles relativas al caso

- Más información: https://blogs.upm.es/ue-upm/

- Contacto: comunidad.microelectronica@upm.es