EBook Gratis

APRENDIZAJE vhdl

Free unaffiliated eBook created from
Stack Overflow contributors.

#vhdl

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vhdl

It is an unofficial and free vhdl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official vhdl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con vhdl

Observaciones

VHDL es un acrónimo compuesto de VHSIC (circuito integrado de muy alta velocidad) HDL (lenguaje de descripción de hardware). Como lenguaje de descripción de hardware, se utiliza principalmente para describir o modelar circuitos. VHDL es un lenguaje ideal para describir circuitos, ya que ofrece construcciones de lenguaje que describen fácilmente el comportamiento concurrente y secuencial junto con un modelo de ejecución que elimina la ambigüedad introducida al modelar el comportamiento concurrente.

VHDL se interpreta normalmente en dos contextos diferentes: para simulación y para síntesis. Cuando se interpreta para la síntesis, el código se convierte (sintetiza) a los elementos de hardware equivalentes que se modelan. Normalmente, solo un subconjunto de VHDL está disponible para su uso durante la síntesis, y las construcciones de lenguaje compatibles no están estandarizadas; Es una función del motor de síntesis utilizado y del dispositivo de hardware de destino. Cuando se interpreta VHDL para simulación, todas las construcciones de lenguaje están disponibles para modelar el comportamiento del hardware.

Versiones

Versión	Fecha de lanzamiento
IEEE 1076-1987	1988-03-31
IEEE 1076-1993	1994-06-06
IEEE 1076-2000	2000-01-30
IEEE 1076-2002	2002-05-17
IEEE 1076c-2007	2007-09-05
IEEE 1076-2008	2009-01-26

Examples

Instalación o configuración

Un programa VHDL puede ser simulado o sintetizado. La simulación es lo que más se parece a la ejecución en otros lenguajes de programación. La síntesis convierte un programa VHDL en una red de puertas lógicas. Muchas herramientas de simulación y síntesis de VHDL forman parte de las suites comerciales de Electronic Design Automation (EDA). Con frecuencia también manejan otros lenguajes de descripción de hardware (HDL), como Verilog, SystemVerilog o SystemC.

Existen algunas aplicaciones gratuitas y de código abierto.

Simulación VHDL

GHDL es probablemente el simulador VHDL libre y de código abierto más maduro. Se presenta en tres sabores diferentes según el backend utilizado: `gcc` , `llvm` o `mcode` . Los siguientes ejemplos muestran cómo usar GHDL (versión `mcode`) y Modelsim, el simulador comercial HDL de Mentor Graphics, bajo un sistema operativo GNU / Linux. Las cosas serían muy similares con otras herramientas y otros sistemas operativos.

Hola Mundo

Crea un archivo `hello_world.vhd` contenga:

```
-- File hello_world.vhd
entity hello_world is
end entity hello_world;

architecture arc of hello_world is
begin
  assert false report "Hello world!" severity note;
end architecture arc;
```

Una unidad de compilación VHDL es un programa completo de VHDL que se puede compilar solo. *Las entidades* son unidades de compilación VHDL que se utilizan para describir la interfaz externa de un circuito digital, es decir, sus puertos de entrada y salida. En nuestro ejemplo, la `entity` se llama `hello_world` y está vacía. El circuito que estamos modelando es una caja negra, no tiene entradas ni salidas. *Las arquitecturas* son otro tipo de unidad compiladora. Siempre están asociados a una `entity` y se utilizan para describir el comportamiento del circuito digital. Una *entidad* puede tener una o más *arquitecturas* para describir el comportamiento de la entidad. En nuestro ejemplo, la *entidad* está asociada solo a una *arquitectura* denominada `arc` que contiene solo una declaración VHDL:

```
  assert false report "Hello world!" severity note;
```

La declaración se ejecutará al comienzo de la simulación e imprimirá el `Hello world!` Mensaje en la salida estándar. La simulación terminará porque no hay nada más que hacer. El archivo fuente VHDL que escribimos contiene dos unidades de compilación. Podríamos haberlos separado en dos archivos diferentes, pero no podríamos haberlos dividido en archivos diferentes: una unidad de compilación debe estar completamente contenida en un archivo fuente. Tenga en cuenta que esta arquitectura no se puede sintetizar porque no describe una función que se pueda traducir directamente a puertas lógicas.

Analiza y ejecuta el programa con GHDL:

```
$ mkdir gh_work
$ ghdl -a --workdir=gh_work hello_world.vhd
```

```
$ ghdl -r --workdir=gh_work hello_world
hello_world.vhd:6:8:@0ms:(assertion note): Hello world!
```

El directorio `gh_work` es donde GHDL almacena los archivos que genera. Esto es lo que dice la opción `--workdir=gh_work` . La fase de análisis verifica la corrección de la sintaxis y produce un archivo de texto que describe las unidades de compilación encontradas en el archivo fuente. La fase de ejecución en realidad compila, vincula y ejecuta el programa. Tenga en cuenta que, en el `mcode` versión de GHDL, no se generan archivos binarios. El programa se recompila cada vez que lo simulamos. Las versiones `gcc` o `llvm` comportan de manera diferente. Tenga en cuenta también que `ghdl -r` no toma el nombre de un archivo fuente VHDL, como lo hace `ghdl -a` , sino el nombre de una unidad de compilación. En nuestro caso le pasamos el nombre de la `entity` . Como solo tiene una `architecture` asociada, no es necesario especificar cuál simular.

Con Modelsim:

```
$ vlib ms_work
$ vmap work ms_work
$ vcom hello_world.vhd
$ vsim -c hello_world -do 'run -all; quit'
...
# ** Note: Hello world!
#    Time: 0 ns  Iteration: 0  Instance: /hello_world
...
```

`vlib` , `vmap` , `vcom` y `vsim` son cuatro comandos que proporciona Modelsim. `vlib` crea un directorio (`ms_work`) donde se almacenarán los archivos generados. `vmap` asocia un directorio creado por `vlib` con un nombre lógico (`work`). `vcom` compila un archivo fuente VHDL y, de forma predeterminada, almacena el resultado en el directorio asociado al nombre lógico de `work` . Finalmente, `vsim` simula el programa y produce el mismo tipo de salida que GHDL. Tenga en cuenta una vez más que lo que pide `vsim` no es un archivo de origen, sino el nombre de una unidad de compilación ya compilada. La opción `-c` le dice al simulador que se ejecute en el modo de línea de comandos en lugar del modo predeterminado de interfaz gráfica de usuario (GUI). La opción `-do` se utiliza para pasar una secuencia de comandos TCL para ejecutar después de cargar el diseño. TCL es un lenguaje de script muy utilizado en las herramientas EDA. El valor de la opción `-do` puede ser el nombre de un archivo o, como en nuestro ejemplo, una cadena de comandos TCL. `run -all; quit` instruye al simulador para que ejecute la simulación hasta que finalice de forma natural, o para siempre si dura para siempre, y luego abandone.

Contador sincrónico

```
-- File counter.vhd
-- The entity is the interface part. It has a name and a set of input / output
-- ports. Ports have a name, a direction and a type. The bit type has only two
-- values: '0' and '1'. It is one of the standard types.
entity counter is
  port(
    clock: in  bit;    -- We are using the rising edge of CLOCK
    reset: in  bit;    -- Synchronous and active HIGH
    data:  out natural -- The current value of the counter
  );
end entity counter;
```

```
-- The architecture describes the internals. It is always associated
-- to an entity.
architecture sync of counter is
  -- The internal signals we use to count. Natural is another standard
  -- type. VHDL is not case sensitive.
  signal current_value: natural;
  signal NEXT_VALUE:    natural;
begin
  -- A process is a concurrent statement. It is an infinite loop.
  process
  begin
    -- The wait statement is a synchronization instruction. We wait
    -- until clock changes and its new value is '1' (a rising edge).
    wait until clock = '1';
    -- Our reset is synchronous: we consider it only at rising edges
    -- of our clock.
    if reset = '1' then
      -- <= is the signal assignment operator.
      current_value <= 0;
    else
      current_value <= next_value;
    end if;
  end process;

  -- Another process. The sensitivity list is another way to express
  -- synchronization constraints. It (approximately) means: wait until
  -- one of the signals in the list changes and then execute the process
  -- body. Sensitivity list and wait statements cannot be used together
  -- in the same process.
  process(current_value)
  begin
    next_value <= current_value + 1;
  end process;

  -- A concurrent signal assignment, which is just a shorthand for the
  -- (trivial) equivalent process.
  data <= current_value;
end architecture sync;
```

Hola Mundo

Hay muchas formas de imprimir el clásico "¡Hola mundo!" mensaje en VHDL. El más simple de todos es probablemente algo como:

```
-- File hello_world.vhd
entity hello_world is
end entity hello_world;

architecture arc of hello_world is
begin
  assert false report "Hello world!" severity note;
end architecture arc;
```

Un entorno de simulación para el contador síncrono.

Entornos de simulación

Un entorno de simulación para un diseño VHDL (el diseño bajo prueba o DUT) es otro diseño VHDL que, como mínimo:

- Declara señales correspondientes a los puertos de entrada y salida del DUT.
- Crea una instancia del DUT y conecta sus puertos a las señales declaradas.
- Crea una instancia de los procesos que conducen las señales conectadas a los puertos de entrada del DUT.

Opcionalmente, un entorno de simulación puede instanciar otros diseños que el DUT, como, por ejemplo, generadores de tráfico en interfaces, monitores para verificar protocolos de comunicación, verificadores automáticos de las salidas del DUT ...

El entorno de simulación es analizado, elaborado y ejecutado. La mayoría de los simuladores ofrecen la posibilidad de seleccionar un conjunto de señales para observar, trazar sus formas de onda gráficas, poner puntos de interrupción en el código fuente, paso en el código fuente ...

Idealmente, un entorno de simulación debería ser utilizable como una prueba robusta de no regresión, es decir, debería detectar automáticamente violaciones de las especificaciones del DUT, reportar mensajes de error útiles y garantizar una cobertura razonable de las funciones del DUT. Cuando dichos entornos de simulación están disponibles, se pueden volver a ejecutar en cada cambio del DUT para verificar que aún es funcionalmente correcto, sin la necesidad de inspecciones visuales tediosas y propensas a errores de las trazas de simulación.

En la práctica, diseñar entornos de simulación ideales o incluso buenos es un desafío. Con frecuencia es tan difícil, o incluso más, que el diseño del propio DUT.

En este ejemplo presentamos un entorno de simulación para el ejemplo del contador síncrono . Mostramos cómo ejecutarlo utilizando GHDL y Modelsim y cómo observar formas de onda gráficas utilizando GTKWave con GHDL y el visor de formas de onda integrado con Modelsim. Luego discutimos un aspecto interesante de las simulaciones: ¿cómo detenerlas?

Un primer entorno de simulación para el contador síncrono.

El contador síncrono tiene dos puertos de entrada y uno puertos de salida. Un entorno de simulación muy simple podría ser:

```
-- File counter_sim.vhd
-- Entities of simulation environments are frequently black boxes without
-- ports.
entity counter_sim is
end entity counter_sim;

architecture sim of counter_sim is
```

```
  -- One signal per port of the DUT. Signals can have the same name as
  -- the corresponding port but they do not need to.
  signal clk:  bit;
  signal rst:  bit;
  signal data: natural;

begin

  -- Instantiation of the DUT
  u0: entity work.counter(sync)
  port map(
    clock => clk,
    reset => rst,
    data  => data
  );

  -- A clock generating process with a 2ns clock period. The process
  -- being an infinite loop, the clock will never stop toggling.
  process
  begin
    clk <= '0';
    wait for 1 ns;
    clk <= '1';
    wait for 1 ns;
  end process;

  -- The process that handles the reset: active from beginning of
  -- simulation until the 5th rising edge of the clock.
  process
  begin
    rst  <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst  <= '0';
    wait; -- Eternal wait. Stops the process forever.
  end process;

end architecture sim;
```

Simulando con GHDL

Vamos a compilar y simular esto con GHDL:

```
$ mkdir gh_work
$ ghdl -a --workdir=gh_work counter_sim.vhd
counter_sim.vhd:27:24: unit "counter" not found in 'library "work"'
counter_sim.vhd:50:35: no declaration for "rising_edge"
```

Entonces los mensajes de error nos dicen dos cosas importantes:

- El analizador GHDL descubrió que nuestro diseño crea una instancia de una entidad llamada `counter` pero esta entidad no se encontró en el `work` biblioteca. Esto es porque no compilamos `counter` antes de `counter_sim` . Al compilar diseños VHDL que crean instancias de entidades, los niveles inferiores siempre deben compilarse antes que los niveles

superiores (los diseños jerárquicos también pueden compilarse de arriba hacia abajo, pero solo si crean instancias de `component` , no entidades).
- La función `rising_edge` utilizada por nuestro diseño no está definida. Esto se debe al hecho de que esta función se introdujo en VHDL 2008 y no le dijimos a GHDL que usara esta versión del lenguaje (por defecto usa VHDL 1993 con tolerancia de sintaxis de VHDL 1987).

Vamos a corregir los dos errores y lanzar la simulación:

```
$ ghdl -a --workdir=gh_work --std=08 counter.vhd counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
^C
```

Tenga en cuenta que la opción `--std=08` es necesaria para el análisis **y la** simulación. Tenga en cuenta también que lanzamos la simulación en la entidad `counter_sim` , architecture `sim` , no en un archivo fuente.

Como nuestro entorno de simulación tiene un proceso que nunca termina (el proceso que genera el reloj), la simulación no se detiene y debemos interrumpirlo manualmente. En su lugar, podemos especificar un tiempo de parada con la opción `--stop-time` :

```
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim --stop-time=60ns
ghdl:info: simulation stopped by --stop-time
```

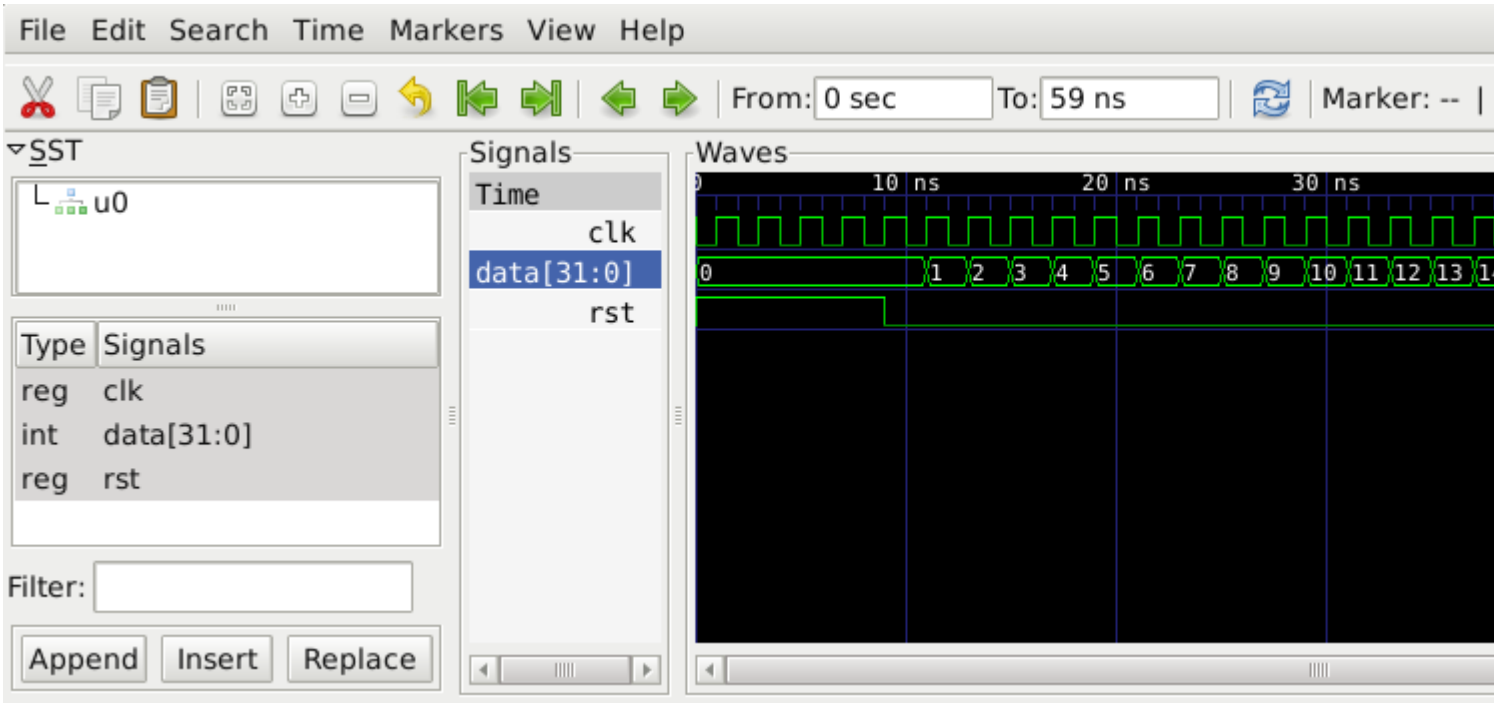
Tal como está, la simulación no nos dice mucho sobre el comportamiento de nuestro DUT. Volcemos los cambios de valor de las señales en un archivo:

```
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim --stop-time=60ns --vcd=counter_sim.vcd
Vcd.Avhpi_Error!
ghdl:info: simulation stopped by --stop-time
```

(ignore el mensaje de error, esto es algo que debe solucionarse en GHDL y eso no tiene ninguna consecuencia). Se ha creado un archivo `counter_sim.vcd` . Contiene en formato VCD (ASCII) todos los cambios de señal durante la simulación. GTKWave puede mostrarnos las formas de onda gráficas correspondientes:

```
$ gtkwave counter_sim.vcd
```

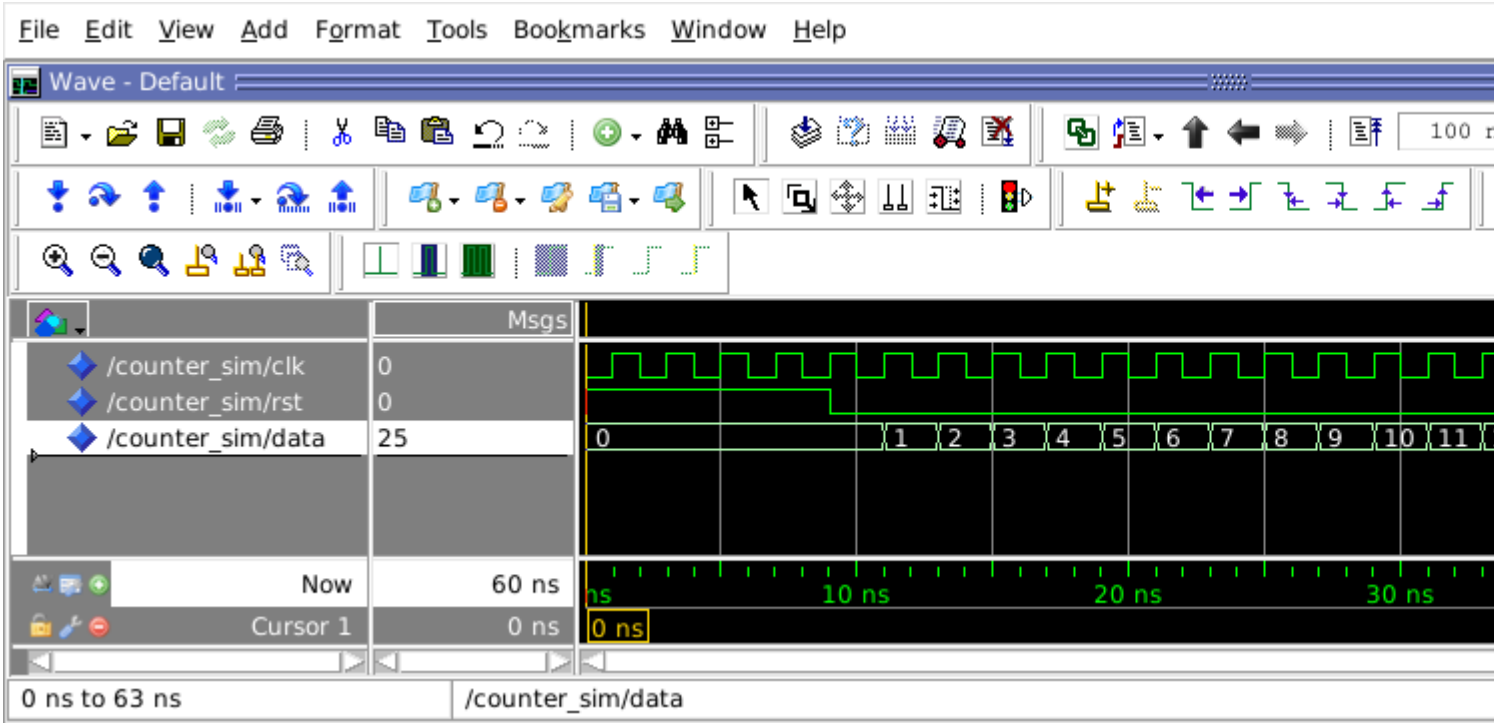
Donde podemos ver que el contador funciona como se espera.



Simulando con Modelsim

El principio es exactamente el mismo con Modelsim:

```
$ vlib ms_work
...
$ vmap work ms_work
...
$ vcom -nologo -quiet -2008 counter.vhd counter_sim.vhd
$ vsim -voptargs="+acc" 'counter_sim(sim)' -do 'add wave /*; run 60ns'
```



Tenga en cuenta que la `-voptargs="+acc"` pasó a `vsim` : impide que el simulador optimice la señal de `data` y nos permite verla en las formas de onda.

Con gracia terminando simulaciones

Con ambos simuladores tuvimos que interrumpir la simulación interminable o especificar un tiempo de parada con una opción dedicada. Esto no es muy conveniente. En muchos casos, el tiempo de finalización de una simulación es difícil de anticipar. Sería mucho mejor detener la simulación desde el interior del código VHDL del entorno de simulación, cuando se alcanza una condición particular, como, por ejemplo, cuando el valor actual del contador llega a 20. Esto se puede lograr con una afirmación en el Proceso que maneja el reinicio:

```
  process
  begin
    rst <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst <= '0';
    loop
      wait until rising_edge(clk);
      assert data /= 20 report "End of simulation" severity failure;
    end loop;
  end process;
```

Mientras los `data` sean diferentes de 20, la simulación continúa. Cuando los `data` llegan a 20, la simulación se bloquea con un mensaje de error:

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
counter_sim.vhd:90:24:@51ns:(assertion failure): End of simulation
ghdl:error: assertion failed
  from: process work.counter_sim(sim2).P1 at counter_sim.vhd:90
ghdl:error: simulation failed
```

Tenga en cuenta que solo hemos vuelto a compilar el entorno de simulación: es el único diseño que ha cambiado y es el nivel superior. Si hubiéramos modificado solo `counter.vhd` , habríamos tenido que volver a compilar ambos: `counter.vhd` porque cambió y `counter_sim.vhd` porque depende de `counter.vhd` .

El bloqueo de la simulación con un mensaje de error no es muy elegante. Incluso puede ser un problema cuando se analizan automáticamente los mensajes de simulación para decidir si una prueba automática de no regresión pasó o no. Una solución mejor y mucho más elegante es detener todos los procesos cuando se alcanza una condición. Esto se puede hacer, por ejemplo, agregando una señal `boolean` fin de simulación (`eof`). Por defecto, se inicializa en `false` al comienzo de la simulación. Uno de nuestros procesos lo establecerá en `true` cuando llegue el momento de finalizar la simulación. Todos los demás procesos monitorearán esta señal y se detendrán con una `wait` eterna cuando se hará `true` :

```
  signal eos:  boolean;
...
  process
  begin
    clk <= '0';
    wait for 1 ns;
    clk <= '1';
    wait for 1 ns;
    if eos then
      report "End of simulation";
      wait;
    end if;
  end process;

  process
  begin
    rst <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst <= '0';
    for i in 1 to 20 loop
      wait until rising_edge(clk);
    end loop;
    eos <= true;
    wait;
  end process;
```

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
counter_sim.vhd:120:24:@50ns:(report note): End of simulation
```

Por último, pero no menos importante, hay una solución aún mejor introducida en VHDL 2008 con el paquete estándar `env` y los procedimientos de `stop` y `finish` que declara:

```
use std.env.all;
...
  process
  begin
    rst    <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst    <= '0';
    for i in 1 to 20 loop
      wait until rising_edge(clk);
    end loop;
    finish;
  end process;
```

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
simulation finished @49ns
```

Señales vs. variables, una breve descripción de la semántica de simulación de VHDL

Este ejemplo trata con uno de los aspectos más fundamentales del lenguaje VHDL: la semántica de simulación. Está dirigido a los principiantes en VHDL y presenta una vista simplificada donde se han omitido muchos detalles (procesos pospuestos, interfaz de procedimiento VHDL, variables compartidas ...) Los lectores interesados en la semántica completa real deben consultar el Manual de referencia de idiomas (LRM).

Señales y variables

La mayoría de los lenguajes de programación imperativos clásicos usan variables. Son contenedores de valor. Un operador de asignación se utiliza para almacenar un valor en una variable:

```
a = 15;
```

y el valor actualmente almacenado en una variable se puede leer y usar en otras declaraciones:

```
if(a == 15) { print "Fifteen" }
```

VHDL también usa variables y tienen exactamente el mismo rol que en la mayoría de los idiomas imperativos. Pero VHDL también ofrece otro tipo de contenedor de valor: la señal. Las señales también almacenan valores, también se pueden asignar y leer. El tipo de valores que se pueden almacenar en señales es (casi) el mismo que en las variables.

Entonces, ¿por qué tener dos tipos de contenedores de valor? La respuesta a esta pregunta es esencial y está en el corazón de la lengua. Comprender la diferencia entre variables y señales es lo primero que debe hacer antes de intentar programar algo en VHDL.

Ilustremos esta diferencia en un ejemplo concreto: el intercambio.

> Nota: todos los siguientes fragmentos de código son parte de procesos. Más adelante veremos qué son los procesos.

```
    tmp := a;
    a   := b;
    b   := tmp;
```

Swaps variables `a` y `b` . Después de ejecutar estas 3 instrucciones, el nuevo contenido de `a` es el contenido antiguo de `b` y viceversa. Como en la mayoría de los lenguajes de programación, se necesita una tercera variable temporal (`tmp`). Si, en lugar de variables, quisiéramos intercambiar señales, escribiríamos:

```
    r <= s;
    s <= r;
```

o:

```
    s <= r;
```

```
    r <= s;
```

¡Con el mismo resultado y sin la necesidad de una tercera señal temporal!

> Nota: el operador de asignación de señal VHDL `<=` es diferente del operador de asignación de variable `:=` .

Veamos un segundo ejemplo en el que suponemos que el subprograma de `print` imprime la representación decimal de su parámetro. Si `a` es una variable entera y su valor actual es 15, ejecutando:

```
    a := 2 * a;
    a := a - 5;
    a := a / 5;
    print(a);
```

imprimirá:

```
5
```

Si ejecutamos este paso a paso en un depurador, podemos ver el valor de `a` cambio de los iniciales 15 a 30, 25 y finalmente 5.

Pero si `s` es una señal entera y su valor actual es 15, ejecutando:

```
    s <= 2 * s;
    s <= s - 5;
    s <= s / 5;
    print(s);
    wait on s;
    print(s);
```

imprimirá:

```
15
3
```

Si ejecutamos este paso a paso en un depurador, no veremos ningún cambio de valor de `s` hasta después de la instrucción de `wait` . ¡Además, el valor final de `s` no será 15, 30, 25 o 5 sino 3!

Este comportamiento aparentemente extraño se debe a la naturaleza fundamentalmente paralela del hardware digital, como veremos en las siguientes secciones.

Paralelismo

VHDL es un lenguaje de descripción de hardware (HDL), es paralelo por naturaleza. Un programa VHDL es una colección de programas secuenciales que se ejecutan en paralelo. Estos programas secuenciales se denominan procesos:

```
P1: process
begin
  instruction1;
  instruction2;
  ...
  instructionN;
end process P1;

P2: process
begin
  ...
end process P2;
```

Los procesos, al igual que el hardware que están modelando, nunca terminan: son bucles infinitos. Después de ejecutar la última instrucción, la ejecución continúa con la primera.

Al igual que con cualquier lenguaje de programación que admite una forma u otra de paralelismo, un programador es responsable de decidir qué proceso ejecutar (y cuándo) durante una simulación VHDL. Además, el lenguaje ofrece construcciones específicas para la comunicación y la sincronización entre procesos.

Programación

El programador mantiene una lista de todos los procesos y, para cada uno de ellos, registra su estado actual que se puede `running` , `run-able` o `suspended` . Hay como máximo un proceso en estado de `running` : el que se ejecuta actualmente. Mientras el proceso actualmente en ejecución no ejecute una instrucción de `wait` , continúa ejecutándose e impide que se ejecute cualquier otro proceso. El programador de VHDL no es preventivo: es responsabilidad de cada proceso suspenderse y dejar que se ejecuten otros procesos. Este es uno de los problemas que los principiantes en VHDL encuentran con frecuencia: el proceso de ejecución libre.

```
  P3: process
    variable a: integer;
  begin
    a := s;
    a := 2 * a;
    r <= a;
  end process P3;
```

> Nota: variable de `a` se declara localmente mientras que las señales `s` y `r` son declarados en otro lugar, en un nivel superior. Las variables VHDL son locales al proceso que las declara y no pueden ser vistas por otros procesos. Otro proceso también podría declarar una variable llamada `a` , no sería la misma variable que la del proceso `P3` .

Tan pronto como el programador reanude el proceso `P3` , la simulación se atascará, la hora actual de la simulación no progresará más y la única forma de detener esto será matar o interrumpir la simulación. La razón es que `P3` no tiene una declaración de `wait` y, por lo tanto, permanecerá en estado de `running` para siempre, repitiendo sus 3 instrucciones. Ningún otro proceso tendrá la oportunidad de ejecutarse, incluso si es `run-able` .

Incluso los procesos que contienen una instrucción de `wait` pueden causar el mismo problema:

```
  P4: process
    variable a: integer;
  begin
    a := s;
    a := 2 * a;
    if a = 16 then
      wait on s;
    end if;
    r <= a;
  end process P4;
```

> Nota: el operador de igualdad VHDL es `=` .

Si el proceso `P4` se reanuda mientras el valor de la señal `s` es 3, se ejecutará para siempre porque la condición a `a = 16` nunca será verdadera.

Supongamos que nuestro programa VHDL no contiene tales procesos patológicos. Cuando el proceso en ejecución ejecuta una instrucción de `wait` , se suspende inmediatamente y el programador la pone en el estado `suspended` . La instrucción de `wait` también conlleva la condición de que el proceso pueda volver a `run-able` . Ejemplo:

```
    wait on s;
```

Me significa *suspender hasta que el valor de la señal* `s` *cambios.* Esta condición es registrada por el planificador. El programador luego selecciona otro proceso entre los `run-able` , lo pone en estado de `running` y lo ejecuta. Y lo mismo se repite hasta que todos `run-able` procesos ejecutables se hayan ejecutado y suspendido.

> **Nota importante:** cuando se pueden `run-able` varios procesos, el estándar VHDL no especifica cómo el programador seleccionará cuál ejecutar. Una consecuencia es que, dependiendo del simulador, la versión del simulador, el sistema operativo o cualquier otra cosa, dos simulaciones del mismo modelo VHDL podrían, en un momento dado, hacer diferentes elecciones y seleccionar un proceso diferente para ejecutar. Si esta elección tuviera un impacto en los resultados de la simulación, podríamos decir que VHDL no es determinista. Como el no determinismo es usualmente indeseable, sería responsabilidad de los programadores evitar situaciones no deterministas. Afortunadamente, VHDL se encarga de esto y es aquí donde las señales entran en escena.

Señales y comunicación entre procesos.

VHDL evita el no determinismo utilizando dos características específicas:

1. **Los procesos pueden intercambiar información solo a través de señales.**

```
  signal r, s: integer;  -- Common to all processes
...
```

```
P5: process
  variable a: integer; -- Different from variable a of process P6
begin
  a := s + 1;
  r <= a;
  a := r + 1;
  wait on s;
end process P5;

P6: process
  variable a: integer; -- Different from variable a of process P5
begin
  a := r + 1;
  s <= a;
  wait on r;
end process P6;
```

Nota: los comentarios de VHDL se extienden desde `--` hasta el final de la línea.

2. **El valor de una señal VHDL no cambia durante la ejecución de procesos**

Cada vez que se asigna una señal, el programador registra el valor asignado, pero el valor actual de la señal permanece sin cambios. Esta es otra diferencia importante con las variables que toman su nuevo valor inmediatamente después de ser asignadas.

Veamos una ejecución del proceso `P5` anterior y supongamos que `a=5` , `s=1 r=0` cuando el planificador lo reanuda. Después de ejecutar la instrucción `a := s + 1;` , el valor de la variable `a` cambia y se convierte en 2 (1 + 1). Al ejecutar la siguiente instrucción `r <= a;` es el nuevo valor de `a` (2) que se asigna a `r` . Pero siendo `r` una señal, el valor actual de `r` sigue siendo 0. Entonces, cuando se ejecuta `a := r + 1;` , la variable `a` toma (inmediatamente) el valor 1 (0 + 1), no 3 (2 + 1) como diría la intuición.

¿Cuándo la señal `r` tomará realmente su nuevo valor? Cuando el programador haya ejecutado todos los procesos ejecutables y todos se suspenderán. Esto también se conoce como: *después de un ciclo* ***delta*** . Solo entonces el programador verá todos los valores que se han asignado a las señales y actualizará realmente los valores de las señales. Una simulación VHDL es una alternancia de fases de ejecución y fases de actualización de señal. Durante las fases de ejecución, el valor de las señales se congela. Simbólicamente, decimos que entre una fase de ejecución y la siguiente fase de actualización de señal transcurrió un *delta* de tiempo. Esto no es en tiempo real. Un ciclo *delta* no tiene duración física.

Gracias a este mecanismo de actualización de señal retardada, VHDL es determinista. Los procesos pueden comunicarse solo con señales y las señales no cambian durante la ejecución de los procesos. Por lo tanto, el orden de ejecución de los procesos no importa: su entorno externo (las señales) no cambia durante la ejecución. `P5.a=5` esto en el ejemplo anterior con los procesos `P5` y `P6` , donde el estado inicial es `P5.a=5` , `P6.a=10` , `s=17` , `r=0` y donde el programador decide ejecutar `P5` primero y `P6` continuación . La siguiente tabla muestra el valor de las dos variables, los valores actuales y siguientes de las señales después de ejecutar cada instrucción de cada proceso:

proceso / instrucción	P5.a	P6.a	s.current	s.next	r.current	r.next
Estado inicial	5	10	17		0	
`P5 / a := s + 1`	18	10	17		0	
`P5 / r <= a`	18	10	17		0	18
`P5 / a := r + 1`	1	10	17		0	18
`P5 / wait on s`	1	10	17		0	18
`P6 / a := r + 1`	1	1	17		0	18
`P6 / s <= a`	1	1	17	1	0	18
`P6 / wait on r`	1	1	17	1	0	18
Después de la actualización de la señal	1	1	1		18	

Con las mismas condiciones iniciales, si el programador decide ejecutar `P6` primero y `P5` continuación:

proceso / instrucción	P5.a	P6.a	s.current	s.next	r.current	r.next
Estado inicial	5	10	17		0	
`P6 / a := r + 1`	5	1	17		0	
`P6 / s <= a`	5	1	17	1	0	
`P6 / wait on r`	5	1	17	1	0	
`P5 / a := s + 1`	18	1	17	1	0	
`P5 / r <= a`	18	1	17	1	0	18
`P5 / a := r + 1`	1	1	17	1	0	18
`P5 / wait on s`	1	1	17	1	0	18
Después de la actualización de la señal	1	1	1		18	

Como podemos ver, después de la ejecución de nuestros dos procesos, el resultado es el mismo, independientemente del orden de ejecución.

Esta semántica de asignación de señal contraintuitiva es la causa de un segundo tipo de problemas que los principiantes de VHDL encuentran con frecuencia: la asignación que aparentemente no funciona porque se retrasa un ciclo delta. Cuando se ejecuta el proceso `P5` paso a paso en un depurador, después de asignar `r` 18 y a `a r + 1` , se puede esperar que el valor

de `a` sea 19, pero el depurador dice obstinadamente que `r=0` y `a=1` ...

> Nota: la misma señal se puede asignar varias veces durante la misma fase de ejecución. En este caso, es la última asignación la que decide el siguiente valor de la señal. Las otras asignaciones no tienen ningún efecto, como si nunca hubieran sido ejecutadas.

Es hora de verificar nuestra comprensión: vuelva a nuestro primer ejemplo de intercambio y trate de entender por qué:

```
process
begin
  ---
  s <= r;
  r <= s;
  ---
end process;
```

en realidad intercambia las señales `r` y `s` sin la necesidad de una tercera señal temporal y por qué:

```
process
begin
  ---
  r <= s;
  s <= r;
  ---
end process;
```

sería estrictamente equivalente. Trate de entender también por qué, si `s` es una señal entera y su valor actual es 15, y ejecutamos:

```
process
begin
  ---
  s <= 2 * s;
  s <= s - 5;
  s <= s / 5;
  print(s);
  wait on s;
  print(s);
  ---
end process;
```

las dos primeras asignaciones de la señal `s` no tienen efecto, por qué `s` se asigna finalmente 3 y por qué los dos valores impresos son 15 y 3.

Tiempo fisico

Para modelar hardware es muy útil poder modelar el tiempo físico tomado por alguna operación. Aquí hay un ejemplo de cómo se puede hacer esto en VHDL. El ejemplo modela un contador

síncrono y es un código VHDL completo y autónomo que se puede compilar y simular:

```
-- File counter.vhd
entity counter is
end entity counter;

architecture arc of counter is
  signal clk: bit; -- Type bit has two values: '0' and '1'
  signal c, nc: natural; -- Natural (non-negative) integers
begin
  P1: process
  begin
    clk <= '0';
    wait for 10 ns; -- Ten nano-seconds delay
    clk <= '1';
    wait for 10 ns; -- Ten nano-seconds delay
  end process P1;

  P2: process
  begin
    if clk = '1' and clk'event then
      c <= nc;
    end if;
    wait on clk;
  end process P2;

  P3: process
  begin
    nc <= c + 1 after 5 ns; -- Five nano-seconds delay
    wait on c;
  end process P3;
end architecture arc;
```

En el proceso `P1` la instrucción de `wait` no se utiliza para esperar hasta que cambie el valor de una señal, como vimos hasta ahora, sino para esperar una duración determinada. Este proceso modela un generador de reloj. Signal `clk` es el reloj de nuestro sistema, es periódico con un período de 20 ns (50 MHz) y tiene un ciclo de trabajo.

Proceso `P2` modelos un registro que, si un borde ascendente de `clk` acaba de ocurrir, asigna el valor de su entrada `nc` a su salida `c` y luego espera para el siguiente cambio de valor de `clk` .

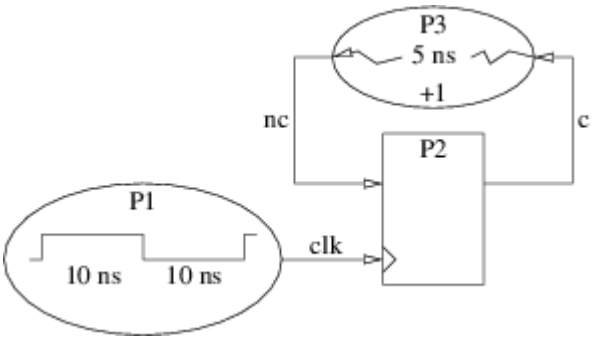
El proceso `P3` modela un incrementador que asigna el valor de su entrada `c` , incrementado en uno, a su salida `nc` ... con un retardo físico de 5 ns. Entonces espera hasta que cambie el valor de su entrada `c` . Esto también es nuevo. Hasta ahora siempre asignamos señales con:

```
  s <= value;
```

que, por las razones explicadas en las secciones anteriores, podemos traducir implícitamente a:

```
  s <= value; -- after delta
```

Este pequeño sistema de hardware digital podría estar representado por la siguiente figura:



Con la introducción del tiempo físico, y sabiendo que también tenemos un tiempo simbólico medido en *delta* , ahora tenemos un tiempo bidimensional que denotaremos `T+D` donde `T` es un tiempo físico medido en nano-segundos y `D` un número de deltas (sin duración física).

La imagen completa

Hay un aspecto importante de la simulación VHDL que aún no discutimos: después de una fase de ejecución, todos los procesos están en estado `suspended` . Informamos de manera informal que el programador luego actualiza los valores de las señales que se han asignado. Pero, en nuestro ejemplo de un contador síncrono, ¿actualizará las señales `clk` , `c` y `nc` al mismo tiempo? ¿Qué pasa con los retrasos físicos? ¿Y qué sucede después con todos los procesos en estado `suspended` y ninguno en estado `run-able` ?

El algoritmo de simulación completo (pero simplificado) es el siguiente:

1. Inicialización
    - Establezca la hora actual `Tc` en 0 + 0 (0 ns, 0 delta-ciclo)
    - Inicializa todas las señales.
    - Ejecute cada proceso hasta que se suspenda en una instrucción de `wait` .
        - Registrar los valores y retrasos de las asignaciones de señales.
        - Registre las condiciones para que se reanude el proceso (retraso o cambio de señal).
    - Calcule la próxima vez `Tn` como la primera de:
        - El tiempo de reanudación de los procesos suspendidos por una `wait for <delay>` .
        - La próxima vez que cambie un valor de señal.

2. Ciclo de simulacion
    - `Tc=Tn` .
    - Actualizar las señales que deben ser.
    - Ponga en estado de `run-able` todos los procesos que esperaban un cambio de valor de una de las señales que se ha actualizado.
    - Ponga en estado de `run-able` todos los procesos que fueron suspendidos por una instrucción de `wait for <delay>` y cuyo tiempo de reanudación es `Tc` .
    - Ejecutar todos los procesos ejecutables hasta que se suspendan.
        - Registrar los valores y retrasos de las asignaciones de señales.
        - Registre las condiciones para que se reanude el proceso (retraso o cambio de

señal).

- Calcule la próxima vez `Tn` como la primera de:
  - El tiempo de reanudación de los procesos suspendidos por una `wait for <delay>` .
  - La próxima vez que cambie un valor de señal.
- Si `Tn` es infinito, detén la simulación. Si no, comienza un nuevo ciclo de simulación.

Simulación manual

Para concluir, ahora ejerzamos manualmente el algoritmo de simulación simplificado en el contador síncrono presentado anteriormente. Decidimos arbitrariamente que, cuando se ejecutan varios procesos, el orden será `P3` > `P2` > `P1` . Las siguientes tablas representan la evolución del estado del sistema durante la inicialización y los primeros ciclos de simulación. Cada señal tiene su propia columna en la que se indica el valor actual. Cuando se ejecuta una asignación de señal, el valor programado se agrega al valor actual, por ejemplo, `a/b@T+D` si el valor actual es `a` y el siguiente valor será `b` en el tiempo `T+D` (tiempo físico más ciclos delta) . Las 3 últimas columnas indican la condición para reanudar los procesos suspendidos (nombre de las señales que deben cambiar o la hora a la que se debe reanudar el proceso).

Fase de inicialización:

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	0 + 0							
Inicializar todas las señales	0 + 0		'0'	0	0			
`P3/nc<=c+1 after 5 ns`	0 + 0		'0'	0	0/1 @ 5 + 0			
`P3/wait on c`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/if clk='1'...`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/end if`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/wait on clk`	0 + 0		'0'	0	0/1 @ 5 + 0		`clk`	`c`
`P1/clk<='0'`	0 + 0		'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0		`clk`	`c`

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
`P1/wait for 10 ns`	0 + 0		'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Calcular la próxima vez	0 + 0	0 + 1	'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`

Ciclo de simulación # 1

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	0 + 1		'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Actualizar señales	0 + 1		'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Calcular la próxima vez	0 + 1	5 + 0	'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`

> Nota: durante el primer ciclo de simulación no hay fase de ejecución porque ninguno de nuestros 3 procesos cumple con su condición de reanudación. `P2` está esperando un cambio de valor de `clk` y ha habido una *transacción* en `clk` , pero como los valores antiguos y nuevos son los mismos, esto no es un *cambio de* valor.

Ciclo de simulación # 2

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	5 + 0		'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Actualizar señales	5 + 0		'0'	0	1	10 + 0	`clk`	`c`
Calcular la próxima vez	5 + 0	10 + 0	'0'	0	1	10 + 0	`clk`	`c`

> Nota: de nuevo, no hay fase de ejecución. `nc` cambió pero ningún proceso está esperando en `nc` .

Ciclo de simulación # 3

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	10 + 0		'0'	0	1	10 + 0	`clk`	`c`
Actualizar señales	10 + 0		'0'	0	1	10 + 0	`clk`	`c`
`P1/clk<='1'`	10 + 0		'0' / '1' @ 10 + 1	0	1		`clk`	`c`

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
P1/wait for 10 ns	10 + 0		'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c
Calcular la próxima vez	10 + 0	10 + 1	'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c

Ciclo de simulación # 4

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	10 + 1		'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c
Actualizar señales	10 + 1		'1'	0	1	20 + 0	clk	c
P2/if clk='1'...	10 + 1		'1'	0	1	20 + 0		c
P2/c<=nc	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0		c
P2/end if	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0		c
P2/wait on clk	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0	clk	c
Calcular la próxima vez	10 + 1	10 + 2	'1'	0/1 @ 10 + 2	1	20 + 0	clk	c

Ciclo de simulación # 5

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	10 + 2		'1'	0/1 @ 10 + 2	1	20 + 0	clk	c
Actualizar señales	10 + 2		'1'	1	1	20 + 0	clk	c
P3/nc<=c+1 after 5 ns	10 + 2		'1'	1	1/2 @ 15 + 0	20 + 0	clk	
P3/wait on c	10 + 2		'1'	1	1/2 @ 15 + 0	20 + 0	clk	c

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Calcular la próxima vez	10 + 2	15 + 0	'1'	1	1/2 @ 15 + 0	20 + 0	clk	c

Nota: se podría pensar que la actualización de `nc` se programaría en `15+2` , mientras que la programamos en `15+0` . Cuando se agrega un retardo físico distinto de cero (aquí `5 ns`) a una hora actual (`10+2`), los ciclos delta se desvanecen. De hecho, los ciclos delta son útiles solo para distinguir diferentes tiempos de simulación `T+0` , `T+1` ... con el mismo tiempo físico `T` Tan pronto como el tiempo físico cambie, los ciclos delta se pueden restablecer.

Ciclo de simulación # 6

Operaciones	Tc	Tn	clk	c	nc	P1	P2	P3
Establecer hora actual	15 + 0		'1'	1	1/2 @ 15 + 0	20 + 0	clk	c
Actualizar señales	15 + 0		'1'	1	2	20 + 0	clk	c
Calcular la próxima vez	15 + 0	20 + 0	'1'	1	2	20 + 0	clk	c

Nota: de nuevo, no hay fase de ejecución. `nc` cambió pero ningún proceso está esperando en `nc` .

Simulación ... cambia a inglés para seguir leyendo.

Lea Empezando con vhdl en línea: https://riptutorial.com/es/vhdl/topic/3803/empezando-con-vhdl

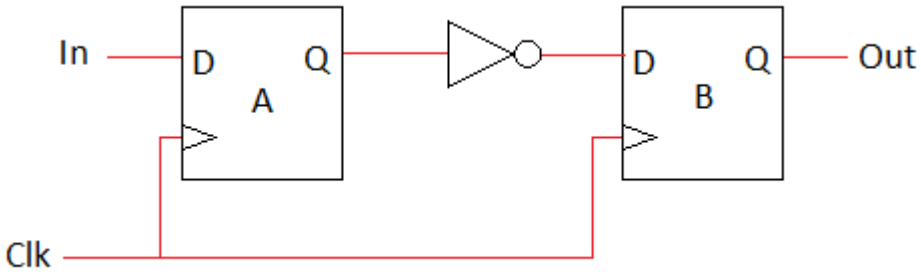
Capítulo 2: Análisis de tiempo estático: ¿qué significa cuando un diseño falla en el tiempo?

Examples

¿Qué es el tiempo?

El concepto de tiempo se relaciona más con la física de los flip flops que con el VHDL, pero es un concepto importante que cualquier diseñador que use VHDL para crear hardware debería saber.

Al diseñar hardware digital, normalmente estamos creando **lógica síncrona** . Esto significa que nuestros datos viajan de flip-flop a flip-flop, posiblemente con alguna lógica combinatoria entre ellos. El diagrama más básico de lógica síncrona que incorpora una función combinatoria se



muestra a continuación:

Un objetivo de diseño importante es la **operación determinista** . En este caso, eso significa que si la salida Q del flop A presentaba la lógica `1` cuando ocurrió el flanco del reloj, esperamos que la salida Q del flop B comience a presentar la lógica `0` cada vez sin excepción.

Con flip-flops **ideales** , como se describe típicamente con VHDL (por ejemplo, `B <= not A when rising_edge(clk);` se supone una operación determinista de `B <= not A when rising_edge(clk);`). Las simulaciones de VHDL conductuales generalmente asumen flip-flops ideales que siempre actúan de manera determinista. Con los flip-flops reales, esto no es tan simple y debemos obedecer **los** requisitos de **configuración** y **retención** correspondientes a cuando cambia la entrada D de un flop para garantizar un funcionamiento confiable.

El tiempo de **configuración** especifica cuánto tiempo debe permanecer sin cambios la entrada D *antes de* la llegada del borde del reloj. El tiempo de **espera** especifica cuánto tiempo debe permanecer sin cambios la entrada D *después de* la llegada del borde del reloj.

Los valores numéricos se basan en la física subyacente de un flip flop y varían significativamente con el **proceso** (imperfecciones en el silicio desde la creación del hardware), **voltaje** (niveles de lógica '0' y '1') y **temperatura** . Normalmente, los valores utilizados para los cálculos son el peor de los casos (el requisito más largo), por lo que podemos garantizar la funcionalidad en cualquier chip y entorno. Los chips se fabrican con rangos permisibles para la alimentación de temperatura

en parte para limitar el peor de los casos que deben considerarse.

La violación de la configuración y los tiempos de espera pueden dar lugar a una variedad de comportamientos no deterministas, incluido el valor lógico incorrecto que aparece en Q, una tensión intermedia que aparece en Q (puede interpretarse como un `0` o `1` por el siguiente elemento lógico), y La salida q oscila. Debido a que todos los números utilizados son valores del caso más desfavorable, las infracciones moderadas generalmente darán como resultado el resultado determinista normal en una pieza específica de hardware, pero una implementación que tenga algún fallo de tiempo no es segura de distribuir en múltiples dispositivos, ya que en el caso real Los valores se acercan al peor de los casos.

Los requisitos típicos para los flip-flops en un FPGA moderno son el tiempo de configuración de 60 pico-segundos, con un requisito de retención de 60 ps. Aunque los detalles de la implementación se dan en un contexto de **FPGA** , casi todo este material se aplica también al diseño de **ASIC** .

Hay varios otros retrasos y valores de tiempo que deben considerarse para determinar si se cumplió el tiempo. Éstos incluyen:

- **Retardo de enrutamiento** : el tiempo que tardan las señales eléctricas en viajar por los cables entre los elementos lógicos
- **Retardo lógico** : el tiempo que tarda la entrada en la lógica combinacional intermedia en afectar la salida. También se conoce comúnmente como retardo de puerta.
- **Retraso de reloj a salida** : otra propiedad física del flip-flop, este es el tiempo que tarda la salida Q en cambiar después de que se produce el borde del reloj.
- **Periodo de reloj** : el tiempo ideal entre dos bordes del reloj. Un período típico para un FPGA moderno que cumple el cronometraje fácilmente es de 5 nano-segundos, pero el diseñador elige el período real utilizado y puede ser moderadamente más corto o drásticamente más largo.
- **Inclinación del reloj** : la diferencia en los retrasos de enrutamiento de la fuente del reloj al flop A y la fuente del reloj al flop B
- **Jitter / incertidumbre del reloj** : una función del ruido eléctrico y los osciladores imperfectos. Esta es la desviación máxima que puede tener el período de reloj del ideal, incorporando tanto un error de frecuencia (ej. El oscilador funciona un 1% demasiado rápido, lo que hace que el período ideal de 5 ns se convierta en 4.95ns con incertidumbre de 50ps) y pico a pico (por ejemplo, el período promedio es de 5 ns, pero 1/1000 ciclos tienen un período de 4,9 ns con 100 ps de jitter)

La verificación de si la implementación del circuito cumple con el tiempo se calcula en dos pasos con dos conjuntos de valores para los retrasos, ya que los retrasos en el peor de los casos para el requisito de retención son los mejores retrasos del caso para el requisito de configuración.

La **verificación de retención** está verificando que el nuevo valor de la salida Q de A en el ciclo de reloj x no llega tan temprano que interrumpe la salida Q de B en el ciclo de reloj x, y por lo tanto no es una función del período de reloj, ya que estamos buscando lo mismo borde del reloj en ambos flops. Cuando falla una verificación de espera, es relativamente fácil de arreglar porque la solución es agregar demora. Las herramientas de implementación pueden aumentar el retraso simplemente agregando más longitud de cable en la ruta.

Para cumplir con el requisito de retención, los retardos de enrutamiento, lógicos y de enrutamiento *más cortos* posibles deben ser acumulativamente más largos que el requisito de retención cuando el sesgo del reloj modifica el requisito de retención.

La **verificación de la configuración** está verificando que el nuevo valor de la salida Q de A en el ciclo x del reloj llega a tiempo para que la salida Q de B lo considere en el ciclo x + 1 del reloj, y por lo tanto es una función del período. Una falla en la verificación de configuración requiere que se retire el retraso o que se aumente el requisito (período de reloj). Las herramientas de implementación no pueden cambiar el período de reloj (que depende del diseñador), y solo hay un gran retraso que puede eliminarse sin cambiar ninguna funcionalidad, por lo que las herramientas no siempre pueden cambiar la ubicación y el enrutamiento de los elementos del circuito para pasar la comprobación de configuración.

Para cumplir con el requisito de configuración, los retrasos de enrutamiento, lógica y enrutamiento *más prolongados* posibles deben ser acumulativamente más cortos que el período de reloj (modificado por sesgo de reloj y fluctuación / incertidumbre) menos el requisito de configuración.

Debido a que se debe conocer el período del reloj (que generalmente se proporciona desde fuera del chip a través de los pines de entrada del reloj) para calcular si se cumplió con la verificación de configuración, todas las herramientas de implementación necesitarán al menos una **restricción de tiempo** proporcionada por el diseñador que indique el período del reloj. La fluctuación de fase / incertidumbre se supone que es 0 o un valor predeterminado pequeño, y los otros valores siempre son conocidos internamente por las herramientas para el FPGA objetivo. Si no se proporciona un período de reloj, la mayoría de las herramientas FPGA verificará la verificación de espera y luego buscará el reloj más rápido que aún permita que todas las rutas se ajusten a la configuración, aunque pasará un tiempo mínimo optimizando las rutas lentas para mejorar el reloj más rápido posible desde la velocidad real necesaria es desconocido.

Si el diseño tiene las restricciones de período requeridas y la lógica no síncrona se excluye adecuadamente del análisis de tiempo (no cubierto en este documento), pero el diseño **aún falla en el tiempo,** hay algunas opciones:

- La opción más simple que no afecta en absoluto a la funcionalidad es **ajustar las directivas** dadas a la herramienta con la esperanza de que probar diferentes estrategias de optimización produzca un resultado que cumpla con el tiempo. Esto no es un éxito confiable, pero a menudo puede encontrar una solución para casos límite.

- El diseñador siempre puede **reducir la frecuencia de reloj** (aumentar el período) para cumplir con las verificaciones de configuración, pero eso tiene sus propias compensaciones funcionales, a saber, que su sistema ha reducido el rendimiento de datos proporcional a la reducción de la velocidad del reloj.

- Los diseños a veces se pueden **refactorizar** para hacer lo mismo con una lógica más simple, o para hacer algo diferente con un resultado final igualmente aceptable para reducir los retrasos combinatorios, lo que facilita las verificaciones de configuración.

- También es una práctica común cambiar el diseño descrito (en el VHDL) a las mismas

operaciones lógicas con el mismo rendimiento pero más latencia al usar más flip-flops y dividir la lógica combinatoria en múltiples ciclos de reloj. Esto se conoce como **canalización** y lleva a una reducción de los retrasos combinatorios (y elimina el retardo de enrutamiento entre lo que antes era múltiples capas de lógica combinatoria). Algunos diseños se prestan bien a la canalización, aunque puede no ser obvio si una ruta lógica larga es una operación monolítica, mientras que otros diseños (como los que incorporan una gran cantidad de **retroalimentación**) no funcionarán en absoluto con la latencia adicional que la tubería conlleva.

Lea Análisis de tiempo estático: ¿qué significa cuando un diseño falla en el tiempo? en línea: https://riptutorial.com/es/vhdl/topic/5936/analisis-de-tiempo-estatico---que-significa-cuando-un-diseno-falla-en-el-tiempo-

Capítulo 3: Comentarios

Introducción

Cualquier lenguaje de programación decente soporta comentarios. En VHDL son especialmente importantes porque comprender un código VHDL, incluso moderadamente sofisticado, es a menudo un desafío.

Examples

Comentarios de una sola línea

Un comentario de una sola línea comienza con dos guiones (`--`) y se extiende hasta el final de la línea. Ejemplo:

```
-- This process models the state register
process(clock, aresetn)
begin
  if aresetn = '0' then         -- Active low, asynchronous reset
    state <= IDLE;
  elsif rising_edge(clock) then -- Synchronized on the rising edge of the clock
    state <= next_state;
  end if;
end process;
```

Comentarios delimitados

A partir de VHDL 2008, un comentario también puede extenderse en varias líneas. Los comentarios de líneas múltiples comienzan con `/*` y terminan con `*/` . Ejemplo:

```
/* This process models the state register.
   It has an active low, asynchronous reset
   and is synchronized on the rising edge
   of the clock. */
process(clock, aresetn)
begin
  if aresetn = '0' then
    state <= IDLE;
  elsif rising_edge(clock) then
    state <= next_state;
  end if;
end process;
```

Los comentarios delimitados también se pueden usar en menos de una línea:

```
-- Finally, we decided to skip the reset...
process(clock/*, aresetn*/)
begin
  /*if aresetn = '0' then
    state <= IDLE;
```

```
  els*/if rising_edge(clock) then
    state <= next_state;
  end if;
end process;
```

Comentarios anidados

Iniciar un nuevo comentario (una sola línea o delimitado) dentro de un comentario (una sola línea o delimitado) no tiene ningún efecto y se ignora. Ejemplos:

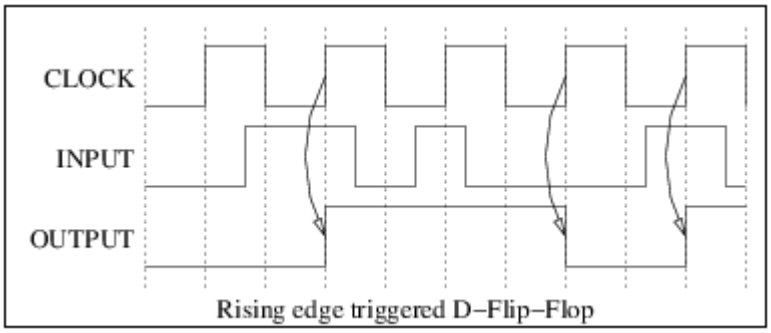
```
-- This is a single-line comment. This second -- has no special meaning.

-- This is a single-line comment. This /* has no special meaning.

/* This is not a
single-line comment.
And this -- has no
special meaning. */

/* This is not a
single-line comment.
And this second /* has no
special meaning. */
```

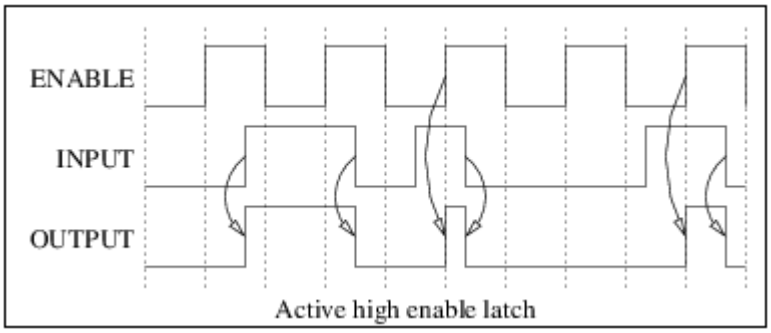
Lea Comentarios en línea: https://riptutorial.com/es/vhdl/topic/9292/comentarios

Capítulo 4: D-Flip-Flops (DFF) y cierres

Observaciones

Los D-Flip-Flops (DFF) y los pestillos son elementos de memoria. Un DFF muestra su entrada en uno u otro borde de su reloj (no en ambos) mientras que un pestillo es transparente en un nivel de su habilitación y memoriza en el otro. La siguiente figura ilustra la diferencia:





Modelar DFFs o pestillos en VHDL es fácil, pero hay algunos aspectos importantes que deben tenerse en cuenta:

- Las diferencias entre los modelos VHDL de DFFs y latches.
- Cómo describir los bordes de una señal.
- Cómo describir conjuntos o reinicios síncronos o asíncronos.

Examples

D-Flip-Flops (DFF)

En todos los ejemplos:

- `clk` es el reloj,
- `d` es la entrada,
- `q` es la salida,
- `srst` es un reinicio alto síncrono activo,
- `srstn` es un reinicio bajo sincronizado activo,

- `arst` es un alto reinicio asíncrono activo,
- `arstn` es un restablecimiento asíncrono bajo activo,
- `sset` es un conjunto alto síncrono activo,
- `ssetn` es un conjunto sincrónico bajo activo,
- `aset` es un conjunto alto asíncrono activo,
- `asetn` es un conjunto asíncrono bajo activo.

Todas las señales son de tipo `ieee.std_logic_1164.std_ulogic` . La sintaxis utilizada es la que lleva a corregir los resultados de síntesis con todos los sintetizadores lógicos. Consulte el ejemplo de *detección del borde* del *reloj* para ver una discusión sobre la sintaxis alternativa.

Reloj de vanguardia

```
process(clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Reloj de borde descendente

```
process(clk)
begin
  if falling_edge(clk) then
    q <= d;
  end if;
end process;
```

Reloj de flanco ascendente, reinicio alto activo síncrono

```
process(clk)
begin
  if rising_edge(clk) then
    if srst = '1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Reloj de flanco ascendente, alto reinicio

activo asíncrono

```
process(clk, arst)
begin
  if arst = '1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Falling edge clock, restablecimiento activo bajo asíncrono, ajuste alto activo síncrono

```
process(clk, arstn)
begin
  if arstn = '0' then
    q <= '0';
  elsif falling_edge(clk) then
    if sset = '1' then
      q <= '1';
    else
      q <= d;
    end if;
  end if;
end process;
```

Reloj de flanco ascendente, reinicio alto activo asíncrono, conjunto bajo activo asíncrono

Nota: el conjunto tiene mayor prioridad que el reinicio

```
process(clk, arst, asetn)
begin
  if asetn = '0' then
    q <= '1';
  elsif arst = '1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Pestillos

En todos los ejemplos:

- `en` está como señal de activación,
- `d` es la entrada,
- `q` es la salida,
- `srst` es un reinicio alto síncrono activo,
- `srstn` es un reinicio bajo sincronizado activo,
- `arst` es un alto reinicio asíncrono activo,
- `arstn` es un restablecimiento asíncrono bajo activo,
- `sset` es un conjunto alto síncrono activo,
- `ssetn` es un conjunto sincrónico bajo activo,
- `aset` es un conjunto alto asíncrono activo,
- `asetn` es un conjunto asíncrono bajo activo.

Todas las señales son de tipo `ieee.std_logic_1164.std_ulogic` . La sintaxis utilizada es la que lleva a corregir los resultados de síntesis con todos los sintetizadores lógicos. Consulte el ejemplo de *detección del borde* del *reloj* para ver una discusión sobre la sintaxis alternativa.

Alta habilitación activa

```
process(en, d)
begin
  if en = '1' then
    q <= d;
  end if;
end process;
```

Activación baja activa

```
process(en, d)
begin
  if en = '0' then
    q <= d;
  end if;
end process;
```

Activación alta activa, reinicio alto síncrono activo

```
process(en, d)
begin
  if en = '1' then
    if srst = '1' then
      q <= '0';
    else
```

```
      q <= d;
    end if;
  end if;
end process;
```

Activación alta activa, reinicio alto asíncrono activo

```
process(en, d, arst)
begin
  if arst = '1' then
    q <= '0';
  elsif en = '1' then
    q <= d;
  end if;
end process;
```

Activación baja activa, reinicio bajo asíncrono activo, ajuste alto activo síncrono

```
process(en, d, arstn)
begin
  if arstn = '0' then
    q <= '0';
  elsif en = '0' then
    if sset = '1' then
      q <= '1';
    else
      q <= d;
    end if;
  end if;
end process;
```

Activación alta activa, reinicio alto activo asíncrono, conjunto bajo activo asíncrono

Nota: el conjunto tiene mayor prioridad que el reinicio

```
process(en, d, arst, asetn)
begin
  if asetn = '0' then
    q <= '1';
  elsif arst = '1' then
    q <= '0';
  elsif en = '1' then
    q <= d;
```

```
  end if;
end process;
```

Detección del borde del reloj

La historia corta

Con VHDL 2008 y si el tipo de reloj es `bit` , `boolean` , `ieee.std_logic_1164.std_ulogic` o `ieee.std_logic_1164.std_logic` , se puede codificar una detección de borde de reloj para borde ascendente

- `if rising_edge(clock) then`
- `if clock'event and clock = '1' then -- type bit, std_ulogic or std_logic`
- `if clock'event and clock then -- type boolean`

y para caer al filo

- `if falling_edge(clock) then`
- `if clock'event and clock = '0' then -- type bit, std_ulogic or std_logic`
- `if clock'event and not clock then -- type boolean`

Esto se comportará como se espera, tanto para la simulación como para la síntesis.

> Nota: la definición de un flanco ascendente en una señal de tipo `std_ulogic` es un poco más compleja que la simple `if clock'event and clock = '1' then` . La función estándar `rising_edge` , por ejemplo, tiene una definición diferente. Incluso si probablemente no hará ninguna diferencia para la síntesis, podría hacer uno para la simulación.

Se recomienda `rising_edge` el uso de las funciones estándar `rising_edge` y `falling_edge` . Con versiones anteriores de VHDL, el uso de estas funciones puede requerir declarar explícitamente el uso de paquetes estándar (por ejemplo, `ieee.numeric_bit` para el tipo de `bit`) o incluso definirlos en un paquete personalizado.

> Nota: no utilice las funciones estándar `rising_edge` y `falling_edge` para detectar bordes de señales que no sean de reloj. Algunos sintetizadores podrían concluir que la señal es un reloj. Sugerencia: la detección de un borde en una señal que no es de reloj se puede hacer frecuentemente muestreando la señal en un registro de desplazamiento y comparando los valores muestreados en diferentes etapas del registro de desplazamiento.

La larga historia

La descripción correcta de la detección de los bordes de una señal de reloj es esencial al modelar D-Flip-Flops (DFF). Un borde es, por definición, una transición de un valor particular a otro. Por ejemplo, podemos definir el flanco ascendente de una señal de tipo `bit` (el tipo enumerado VHDL estándar que toma dos valores: `'0'` y `'1'`) como la transición de `'0'` a `'1'` . Para el tipo `boolean` podemos definirlo como una transición de `false` a `true` .

Con frecuencia, se utilizan tipos más complejos. El tipo `ieee.std_logic_1164.std_ulogic` , por ejemplo, también es un tipo enumerado, como `bit` o `boolean` , pero tiene 9 valores en lugar de 2:

Valor	Sentido
`'U'`	Sin inicializar
`'X'`	Forzando desconocido
`'0'`	Forzando bajo nivel
`'1'`	Forzando alto nivel
`'Z'`	Alta impedancia
`'W'`	Débil desconocido
`'L'`	Débil bajo nivel
`'H'`	Débil alto nivel
`'-'`	No importa

Definir un flanco ascendente en un tipo de este tipo es un poco más complejo que para `bit` o `boolean` . Podemos, por ejemplo, decidir que es una transición de `'0'` a `'1'` . Pero también podemos decidir que es una transición de `'0'` o `'L'` a `'1'` o `'H'` .

> Nota: esta es la segunda definición que usa el estándar para la función `rising_edge(signal s: std_ulogic)` definida en `ieee.std_logic_1164` .

Cuando se analizan las diversas formas de detectar bordes, es importante considerar el tipo de señal. También es importante tener en cuenta el objetivo de modelado: ¿solo simulación o síntesis lógica? Vamos a ilustrar esto con algunos ejemplos:

Flanco DFF con bit de tipo

```
signal clock, d, q: bit;
...
P1: process(clock)
begin
  if clock = '1' then
    q <= d;
  end if;
end process P1;
```

Técnicamente, en un punto de vista semántico de simulación puro, el proceso `P1` modela un DFF de flanco ascendente. De hecho, la asignación `q <= d` se ejecuta si y solo si:

- `clock` cambiado (esto es lo que expresa la lista de sensibilidad) **y**
- El valor actual del `clock` es `'1'` .

Como el `clock` es de tipo bit y tipo bit solo tiene valores `'0'` y `'1'` , esto es exactamente lo que definimos como un flanco ascendente de una señal de tipo bit. Cualquier simulador manejará este modelo como esperamos.

> Nota: para los sintetizadores lógicos, las cosas son un poco más complejas, como veremos más adelante.

DFF de flanco ascendente con restablecimiento alto activo asíncrono y bit de tipo

Para agregar un restablecimiento alto activo asíncrono a nuestro DFF, se podría intentar algo como:

```
signal clock, reset, d, q: bit;
...
P2_BOGUS: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif clock = '1' then
    q <= d;
  end if;
end process P2_BOGUS;
```

Pero **esto no funciona** . La condición para que se ejecute la asignación `q <= d` debe ser: *un flanco ascendente del `clock` mientras se `reset = '0'`* . Pero lo que modelamos es:

- `clock` o `reset` o ambos cambiados **y**
- `reset = '0'` **y**
- `clock = '1'`

Que no es lo mismo: si `reset` cambia de `'1'` a `'0'` , mientras que `clock = '1'` la asignación se ejecutarán si bien **no** es un flanco ascendente de `clock` .

De hecho, no hay forma de modelar esto en VHDL sin la ayuda de un atributo de señal:

```
P2_OK: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif clock = '1' and clock'event then
    q <= d;
  end if;
end process P2_OK;
```

El `clock'event` es el `event` atributo de señal aplicado al `clock` señal. Se evalúa como un valor `boolean` y es `true` si y solo si el `clock` señal cambió durante la fase de actualización de la señal que precedió a la fase de ejecución actual. Gracias a esto, el proceso `P2_OK` ahora modela perfectamente lo que queremos en simulación (y síntesis).

Semántica de síntesis

Muchos sintetizadores lógicos identifican detecciones de borde de señal basadas en patrones sintácticos, no en la semántica del modelo VHDL. En otras palabras, consideran cómo se ve el código VHDL, no qué comportamiento modela. Uno de los patrones que todos reconocen es:

```
if clock = '1' and clock'event then
```

Entonces, incluso en el ejemplo del proceso `P1` , deberíamos usarlo si queremos que nuestro sintetizador sea sintetizable por todos los sintetizadores lógicos:

```
signal clock, d, q: bit;
...
P1_OK: process(clock)
begin
  if clock = '1' and clock'event then
    q <= d;
  end if;
end process P1_OK;
```

La parte de la condición de " `and clock'event` es completamente redundante con la lista de sensibilidad, pero como algunos sintetizadores lo necesitan ...

DFF de flanco ascendente con restablecimiento alto activo asíncrono y tipo `std_ulogic`

En este caso, la expresión del flanco ascendente del reloj y la condición de restablecimiento puede complicarse. Si mantenemos la definición de un flanco ascendente que propusimos anteriormente y si consideramos que el restablecimiento está activo si es `'1'` o `'H'` , el modelo se convierte en:

```
library ieee;
use ieee.std_logic_1164.all;
...
signal clock, reset, d, q: std_ulogic;
...
P4: process(clock, reset)
begin
  if reset = '1' or reset = 'H' then
    q <= '0';
  elsif clock'event and
        (clock'last_value = '0' or clock'last_value = 'L') and
        (clock = '1' or clock = 'H') then
    q <= d;
  end if;
end process P4;
```

> Nota: `'last_value` es otro atributo de señal que devuelve el valor que tenía la señal antes del último cambio de valor.

Funciones de ayuda

El estándar VHDL 2008 ofrece varias funciones de ayuda para simplificar la detección de bordes de señal, especialmente con tipos enumerados de valores múltiples como `std_ulogic` . El paquete `std.standard` define las funciones `rising_edge` y `falling_edge` en los tipos `bit` y `boolean` y el paquete `ieee.std_logic_1164` las define en los tipos `std_ulogic` y `std_logic` .

> Nota: con versiones anteriores de VHDL, el uso de estas funciones puede requerir declarar explícitamente el uso de paquetes estándar (por ejemplo, ieee.numeric_bit para el tipo de bit) o incluso definirlos en un paquete de usuario.

Revisemos los ejemplos anteriores y usemos las funciones de ayuda:

```
signal clock, d, q: bit;
...
P1_OK_NEW: process(clock)
begin
  if rising_edge(clock) then
    q <= d;
  end if;
end process P1_OK_NEW;
```

```
signal clock, d, q: bit;
...
P2_OK_NEW: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process P2_OK_NEW;
```

```
library ieee;
use ieee.std_logic_1164.all;
...
signal clock, reset, d, q: std_ulogic;
...
P4_NEW: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process P4_NEW;
```

> Nota: en este último ejemplo también simplificamos la prueba en el reinicio. Los reinicios flotantes, de alta impedancia son bastante raros y, en la mayoría de los casos, esta versión simplificada funciona para simulación y síntesis.

Lea D-Flip-Flops (DFF) y cierres en línea: https://riptutorial.com/es/vhdl/topic/5983/d-flip-flops--dff--y-cierres

Capítulo 5: Diseño de hardware digital utilizando VHDL en pocas palabras

Introducción

En este tema, proponemos un método simple para diseñar correctamente circuitos digitales simples con VHDL. El método se basa en diagramas de bloques gráficos y en un principio fácil de recordar:

Piense en el hardware primero, codifique VHDL a continuación

Está dirigido a los principiantes en el diseño de hardware digital utilizando VHDL, con un conocimiento limitado de la semántica de síntesis del lenguaje.

Observaciones

El diseño de hardware digital con VHDL es simple, incluso para principiantes, pero hay algunas cosas importantes que debe saber y un pequeño conjunto de reglas que debe obedecer. La herramienta utilizada para transformar una descripción de VHDL en hardware digital es un sintetizador lógico. La semántica del lenguaje VHDL utilizado por los sintetizadores lógicos es bastante diferente de la semántica de simulación descrita en el Manual de referencia del lenguaje (LRM). Peor aún: no está estandarizado y varía entre las herramientas de síntesis.

El método propuesto introduce varias limitaciones importantes en aras de la simplicidad:

- Sin pestillos disparados por nivel.
- Los circuitos están sincronizados en el borde ascendente de un solo reloj.
- No hay reinicio ni ajuste asíncrono.
- No hay unidad múltiple en señales resueltas.

El ejemplo del diagrama de bloques , primero de una serie de 3, presenta brevemente los conceptos básicos del hardware digital y propone una breve lista de reglas para diseñar un diagrama de bloques de un circuito digital. Las reglas ayudan a garantizar una traducción directa al código VHDL que simula y sintetiza como se espera.

El ejemplo de codificación explica la traducción de un diagrama de bloques al código VHDL y lo ilustra en un circuito digital simple.

Finalmente, el ejemplo del concurso de diseño de John Cooley muestra cómo aplicar el método propuesto en un ejemplo más complejo de circuito digital. También elabora las limitaciones introducidas y relaja algunas de ellas.

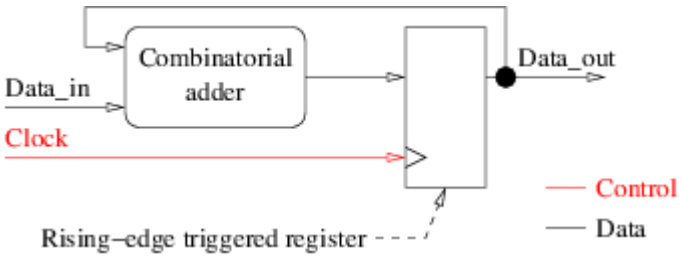
Examples

Diagrama de bloques

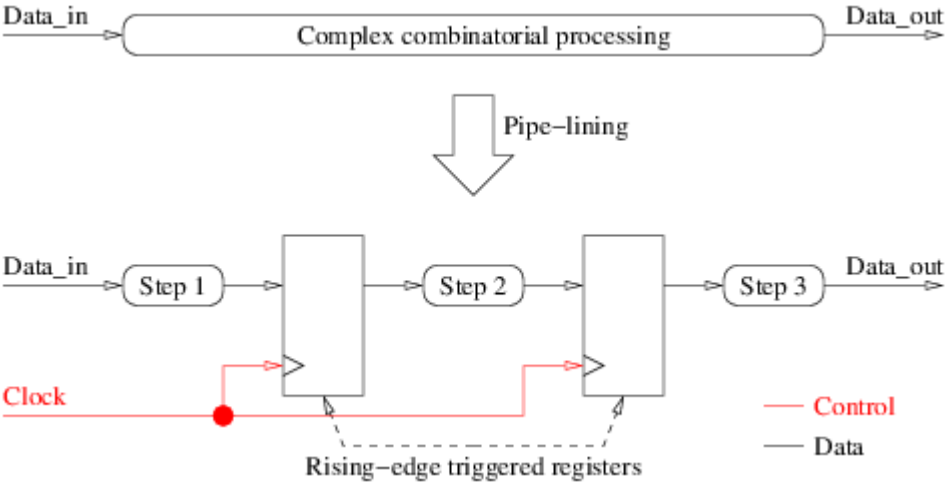
El hardware digital se construye a partir de dos tipos de primitivas de hardware:

- Puertas combinatorias (inversores, y, o xor, sumadores completos de 1 bit, multiplexores de 1 bit ...) Estas puertas lógicas realizan un cálculo booleano simple en sus entradas y producen una salida. Cada vez que cambia una de sus entradas, comienzan a propagar señales eléctricas y, después de un breve retraso, la salida se estabiliza al valor resultante. El retardo de propagación es importante porque está fuertemente relacionado con la velocidad a la que puede funcionar el circuito digital, es decir, su frecuencia de reloj máxima.
- Elementos de memoria (latches, D-flip-flops, RAMs ...). Contrariamente a las puertas lógicas combinatorias, los elementos de memoria no reaccionan inmediatamente al cambio de cualquiera de sus entradas. Tienen entradas de datos, entradas de control y salidas de datos. Reaccionan ante una combinación particular de entradas de control, no ante ningún cambio de sus entradas de datos. El D-flip-flop disparado por el flanco ascendente (DFF), por ejemplo, tiene una entrada de reloj y una entrada de datos. En cada flanco ascendente del reloj, la entrada de datos se muestrea y se copia en la salida de datos que permanece estable hasta el siguiente flanco ascendente del reloj, incluso si la entrada de datos cambia entre ellos.

Un circuito de hardware digital es una combinación de lógica combinatoria y elementos de memoria. Los elementos de memoria tienen varios roles. Uno de ellos es permitir la reutilización de la misma lógica combinatoria para varias operaciones consecutivas en diferentes datos. Los circuitos que usan esto a menudo se denominan *circuitos secuenciales* . La siguiente figura muestra un ejemplo de un circuito secuencial que acumula valores enteros utilizando el mismo sumador combinatorio, gracias a un registro activado de flanco ascendente. También es nuestro primer ejemplo de diagrama de bloques.



El revestimiento de tuberías es otro uso común de los elementos de memoria y la base de muchas arquitecturas de microprocesadores. Su objetivo es aumentar la frecuencia de reloj de un circuito dividiendo un procesamiento complejo en una sucesión de operaciones más simples y paralelizando la ejecución de varios procesamientos consecutivos:



El diagrama de bloques es una representación gráfica del circuito digital. Ayuda a tomar las decisiones correctas y obtener una buena comprensión de la estructura general antes de la codificación. Es el equivalente de las fases de análisis preliminares recomendadas en muchos métodos de diseño de software. Los diseñadores experimentados frecuentemente saltan esta fase de diseño, al menos para circuitos simples. Sin embargo, si usted es un principiante en diseño de hardware digital, y si desea codificar un circuito digital en VHDL, adoptar las 10 reglas simples a continuación para dibujar su diagrama de bloques debería ayudarlo a hacerlo bien:

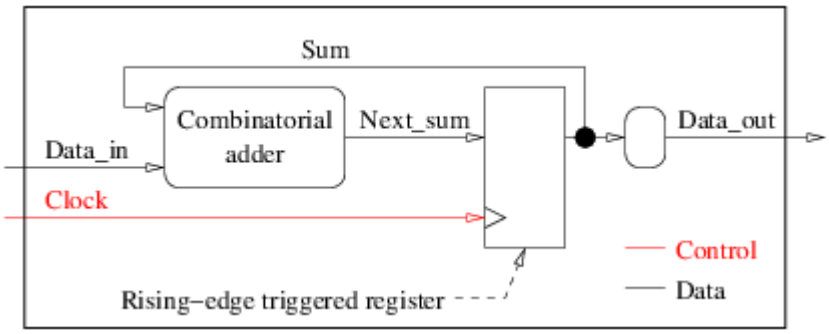
1. Rodea tu dibujo con un gran rectángulo. Este es el límite de tu circuito. Todo lo que cruza este límite es un puerto de entrada o salida. La entidad VHDL describirá este límite.
2. Claramente, los registros activados por flanco (por ejemplo, bloques cuadrados) de la lógica combinatoria (por ejemplo, bloques redondos). En VHDL se traducirán en procesos pero de dos tipos muy diferentes: sincrónicos y combinatorios.
3. No utilice pestillos activados por nivel, use solo registros activados por flanco ascendente. Esta restricción no proviene de VHDL, que es perfectamente utilizable para modelar pestillos. Es solo un consejo razonable para principiantes. Los cierres son menos necesarios y su uso plantea muchos problemas que probablemente deberíamos evitar, al menos para nuestros primeros diseños.
4. Utilice el mismo reloj único para todos sus registros activados de flanco ascendente. Una vez más, esta restricción está aquí por simplicidad. No proviene de VHDL, que es perfectamente utilizable para modelar sistemas de reloj múltiple. Nombra el reloj del `clock` . Viene del exterior y es una entrada de todos los bloques cuadrados y solo de ellos. Si lo desea, ni siquiera represente el reloj, es el mismo para todos los bloques cuadrados y puede dejarlo implícito en su diagrama.
5. Representa las comunicaciones entre bloques con flechas con nombre y orientadas. Para el bloque del que proviene una flecha, la flecha es una salida. Para el bloque al que va una flecha, la flecha es una entrada. Todas estas flechas se convertirán en puertos de la entidad VHDL, si están cruzando el rectángulo grande, o señales de la arquitectura VHDL.
6. Las flechas tienen un solo origen, pero pueden tener varios destinos. De hecho, si una flecha tuviera varios orígenes, crearíamos una señal VHDL con varios controladores. Esto no es completamente imposible, pero requiere un cuidado especial para evitar cortocircuitos. Así evitaremos esto por ahora. Si una flecha tiene varios destinos, bifurque la flecha tantas veces como sea necesario. Use puntos para distinguir los cruces conectados y no conectados.

7. Algunas flechas vienen de fuera del rectángulo grande. Estos son los puertos de entrada de la entidad. Una flecha de entrada no puede ser también la salida de ninguno de sus bloques. Esto se aplica mediante el lenguaje VHDL: los puertos de entrada de una entidad se pueden leer pero no se pueden escribir. Esto es de nuevo para evitar cortocircuitos.
8. Algunas flechas salen afuera. Estos son los puertos de salida. En las versiones VHDL anteriores a 2008, los puertos de salida de una entidad se pueden escribir pero no leer. Por lo tanto, una flecha de salida debe tener un solo origen y un único destino: el exterior. Sin bifurcaciones en las flechas de salida, una flecha de salida no puede ser también la entrada de uno de sus bloques. Si desea usar una flecha de salida como entrada para algunos de sus bloques, inserte un nuevo bloque redondo para dividirlo en dos partes: la interna, con tantas horquillas como desee, y la flecha de salida que proviene de la nueva Bloquea y sale a la calle. El nuevo bloque se convertirá en una simple asignación continua en VHDL. Una especie de cambio de nombre transparente. Desde VHDL 2008 también se pueden leer los puertos de salida.
9. Todas las flechas que no vienen o van desde / hacia el exterior son señales internas. Los declararás todos en la arquitectura VHDL.
10. Cada ciclo en el diagrama debe comprender al menos un bloque cuadrado. Esto no se debe a VHDL. Viene de los principios básicos del diseño de hardware digital. Los bucles combinatorios deben ser absolutamente evitados. Excepto en casos muy raros, no producen ningún resultado útil. Y un ciclo del diagrama de bloques que comprendería solo bloques redondos sería un bucle combinatorio.

No olvide revisar cuidadosamente la última regla, es tan esencial como las otras, pero puede ser un poco más difícil de verificar.

A menos que necesite absolutamente características que excluimos por ahora, como pestillos, relojes múltiples o señales con múltiples controladores, debe dibujar fácilmente un diagrama de bloques de su circuito que cumpla con las 10 reglas. De lo contrario, es probable que el problema sea con el circuito que desea, no con VHDL o el sintetizador lógico. Y probablemente significa que el circuito que desea **no** es hardware digital.

La aplicación de las 10 reglas a nuestro ejemplo de un circuito secuencial llevaría a un diagrama de bloques como:



1. El rectángulo grande alrededor del diagrama está cruzado por 3 flechas, que representan los puertos de entrada y salida de la entidad VHDL.
2. El diagrama de bloques tiene dos bloques redondos (combinatorios), el sumador y el bloque de cambio de nombre de salida, y un bloque cuadrado (síncrono), el registro.
3. Utiliza solo registros activados por flanco.

4. Solo hay un reloj, llamado `clock` y solo usamos su flanco ascendente.
5. El diagrama de bloques tiene cinco flechas, una con un tenedor. Corresponden a dos señales internas, dos puertos de entrada y un puerto de salida.
6. Todas las flechas tienen un origen y un destino, excepto la flecha denominada `Sum` que tiene dos destinos.
7. Las flechas `Data_in` y `Clock` son nuestros dos puertos de entrada. No son salida de nuestros propios bloques.
8. La flecha `Data_out` es nuestro puerto de salida. Para ser compatibles con las versiones VHDL anteriores a 2008, agregamos un bloque de redenominación (redondeo) adicional entre `Sum` y `Data_out` . Por lo tanto, `Data_out` tiene exactamente una fuente y un destino.
9. `Sum` y `Next_sum` son nuestras dos señales internas.
10. Hay exactamente un ciclo en la gráfica y comprende un bloque cuadrado.

Nuestro diagrama de bloques cumple con las 10 reglas. El ejemplo de codificación detallará cómo traducir este tipo de diagramas de bloques en VHDL.

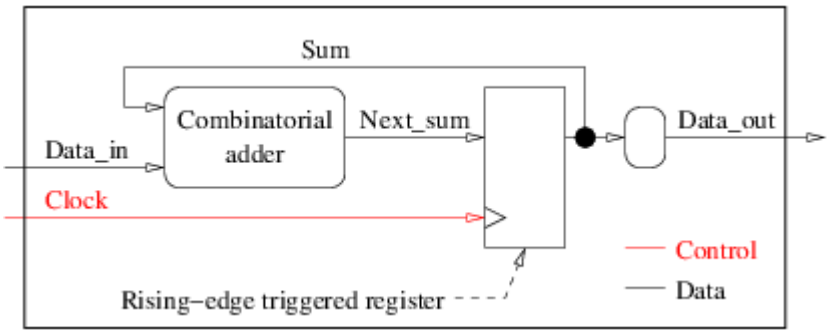
Codificación

Este ejemplo es el segundo de una serie de 3. Si aún no lo hizo, primero lea el ejemplo del diagrama de bloques .

Con un diagrama de bloques que cumple con las 10 reglas (consulte el ejemplo del diagrama de bloques), la codificación VHDL se vuelve sencilla:

- el gran rectángulo circundante se convierte en la entidad VHDL,
- Las flechas internas se convierten en señales VHDL y se declaran en la arquitectura.
- Cada bloque cuadrado se convierte en un proceso síncrono en el cuerpo de la arquitectura.
- Cada bloque redondo se convierte en un proceso combinatorio en el cuerpo de la arquitectura.

Ilustrémoslo en el diagrama de bloques de un circuito secuencial:



El modelo VHDL de un circuito comprende dos unidades de compilación:

- La entidad que describe el nombre del circuito y su interfaz (nombres de puertos, direcciones y tipos). Es una traducción directa del gran rectángulo circundante del diagrama de bloques. Suponiendo que los datos son enteros y que el `clock` usa el `bit` tipo VHDL (solo dos valores: `'0'` y `'1'`), la entidad de nuestro circuito secuencial podría ser:

```
entity sequential_circuit is
  port(
    Data_in:  in  integer;
    Clock:    in  bit;
    Data_out: out integer
  );
end entity sequential_circuit;
```

- La arquitectura que describe las partes internas del circuito (lo que hace). Aquí es donde se declaran las señales internas y donde se instancian todos los procesos. El esqueleto de la arquitectura de nuestro circuito secuencial podría ser:

```
architecture ten_rules of sequential_circuit is
  signal Sum, Next_sum: integer;
begin
  <...processes...>
end architecture ten_rules;
```

Tenemos tres procesos para agregar al cuerpo de la arquitectura, uno síncrono (bloque cuadrado) y dos combinatorios (bloques redondos).

Un proceso síncrono se ve así:

```
process(clock)
begin
  if rising_edge(clock) then
    o1 <= i1;
    ...
    ox <= ix;
  end if;
end process;
```

donde `i1, i2,..., ix` son **todas las** flechas que entran en el bloque cuadrado correspondiente del diagrama y `o1, ..., ox` son **todas las** flechas que `o1, ..., ox` el bloque cuadrado correspondiente del diagrama. Absolutamente nada será cambiado, excepto los nombres de las señales, por supuesto. Nada. Ni siquiera un solo personaje.

El proceso síncrono de nuestro ejemplo es así:

```
  process(clock)
  begin
    if rising_edge(clock) then
      Sum <= Next_sum;
    end if;
  end process;
```

Que se puede traducir de manera informal a: si el `clock` cambia, y solo entonces, si el cambio es un flanco ascendente (`'0'` a `'1'`), asigne el valor de la señal `Next_sum` a la señal `Sum` .

Un proceso combinatorio se ve así:

```
process(i1, i2,... , ix)
  variable v1: <type_of_v1>;
```

```
  ...
  variable vy: <type_of_vy>;
begin
  v1 := <default_value_for_v1>;
  ...
  vy := <default_value_for_vy>;
  o1 <= <default_value_for_o1>;
  ...
  oz <= <default_value_for_oz>;
  <statements>
end process;
```

donde `i1, i2,..., in` son **todas las** flechas que entran en el bloque redondo correspondiente del diagrama. **todo** y no mas No olvidaremos ninguna flecha y no agregaremos nada más a la lista.

`v1, ..., vy` son variables que podemos necesitar para simplificar el código del proceso. Tienen exactamente el mismo rol que en cualquier otro lenguaje de programación imperativo: mantener valores temporales. Deben ser absolutamente asignados todos antes de ser leído. Si no garantizamos esto, el proceso ya no será combinatorio, ya que modelará el tipo de elementos de memoria para retener el valor de algunas variables de una ejecución de proceso a la siguiente. Este es el motivo de las declaraciones `vi := <default_value_for_vi>` al comienzo del proceso. Tenga en cuenta que `<default_value_for_vi>` deben ser constantes. Si no, si son expresiones, podríamos usar accidentalmente variables en las expresiones y leer una variable antes de asignarla.

`o1, ..., om` son **todas las** flechas que `o1, ..., om` el bloque redondo correspondiente de su diagrama. **todo** y no mas Absolutamente deben ser asignados al menos una vez durante la ejecución del proceso. Como las estructuras de control VHDL (`if case` ...) pueden evitar fácilmente que se asigne una señal de salida, le recomendamos encarecidamente que asigne a cada una de ellas, incondicionalmente, con un valor constante `<default_value_for_oi>` al comienzo del proceso. De esta manera, incluso si una instrucción `if` enmascara una asignación de señal, de todos modos habrá recibido un valor.

Absolutamente nada se cambiará a este esqueleto VHDL, excepto los nombres de las variables, si las hay, los nombres de las entradas, los nombres de las salidas, los valores de `<default_value_for_..>` constants y `<statements>` . **No** se olvide de una única asignación de valores por defecto, si lo hace la síntesis inferirá elementos de memoria no deseadas (lo más probable pestillos) y el resultado no será lo que inicialmente quería.

En nuestro ejemplo de circuito secuencial, el proceso de sumador combinatorio es:

```
  process(Sum, Data_in)
  begin
    Next_sum <= 0;
    Next_sum <= Sum + Data_in;
  end process;
```

Lo que se puede traducir de manera informal a: si `Sum` o `Data_in` (o ambos) cambian, asigne el valor 0 para señalar a `Next_sum` y luego asigne nuevamente el valor `Sum + Data_in` .

Como la primera asignación (con el valor predeterminado constante `0`) es seguida

inmediatamente por otra asignación que la sobrescribe, podemos simplificar:

```
  process(Sum, Data_in)
  begin
    Next_sum <= Sum + Data_in;
  end process;
```

El segundo proceso combinatorio corresponde al bloque redondo que agregamos en una flecha de salida con más de un destino para cumplir con las versiones VHDL anteriores a 2008. Su código es simplemente:

```
  process(Sum)
  begin
    Data_out <= 0;
    Data_out <= Sum;
  end process;
```

Por la misma razón que con el otro proceso combinatorio, podemos simplificarlo como:

```
  process(Sum)
  begin
    Data_out <= Sum;
  end process;
```

El código completo para el circuito secuencial es:

```
-- File sequential_circuit.vhd
entity sequential_circuit is
  port(
    Data_in:  in  integer;
    Clock:    in  bit;
    Data_out: out integer
  );
end entity sequential_circuit;

architecture ten_rules of sequential_circuit is
  signal Sum, Next_sum: integer;
begin
  process(clock)
  begin
    if rising_edge(clock) then
      Sum <= Next_sum;
    end if;
  end process;

  process(Sum, Data_in)
  begin
    Next_sum <= Sum + Data_in;
  end process;

  process(Sum)
  begin
    Data_out <= Sum;
  end process;
end architecture ten_rules;
```

Nota: podríamos escribir los tres procesos en cualquier orden, no cambiaríamos nada al resultado final en la simulación o en la síntesis. Esto se debe a que los tres procesos son declaraciones simultáneas y VHDL los trata como si fueran realmente paralelos.

Concurso de diseño de John Cooley.

Este ejemplo se deriva directamente del concurso de diseño de John Cooley en SNUG'95 (reunión del grupo de usuarios de Synopsys). El concurso estaba destinado a oponerse a los diseñadores de VHDL y Verilog en el mismo problema de diseño. Lo que John tenía en mente era probablemente determinar qué idioma era el más eficiente. Los resultados fueron que 8 de los 9 diseñadores de Verilog lograron completar el concurso de diseño, pero ninguno de los 5 diseñadores de VHDL pudo. Con suerte, usando el método propuesto, haremos un trabajo mucho mejor.

Presupuesto

Nuestro objetivo es diseñar en VHDL (entidad y arquitectura) sintetizables, un contador síncrono up-by-down, down-by-5, cargable, módulo 512, con salida de acarreo, salida de préstamo y salida de paridad. El contador es un contador sin signo de 9 bits, por lo que oscila entre 0 y 511. La especificación de la interfaz del contador se muestra en la siguiente tabla:

Nombre	Ancho de bits	Dirección	Descripción
RELOJ	1	Entrada	Reloj maestro; El contador está sincronizado en el flanco ascendente de CLOCK.
DI	9	Entrada	Bus de entrada de datos; el contador está cargado con DI cuando UP y DOWN son bajos
ARRIBA	1	Entrada	Comando de conteo hasta por 3; cuando UP es alto y DOWN es bajo, el contador se incrementa en 3, envolviendo su valor máximo (511)
ABAJO	1	Entrada	Comando de conteo de por 5; cuando ABAJO es alto y ARRIBA es bajo, el contador disminuye en 5, envolviendo su valor mínimo (0)
CO	1	Salida	Llevar a cabo la señal; alto solo cuando se cuenta más allá del valor máximo (511) y, por lo tanto, se ajusta
BO	1	Salida	Pedir prestado la señal; alto solo cuando se cuenta por debajo del valor mínimo (0) y, por lo tanto, se ajusta
HACER	9	Salida	Bus de salida; el valor actual del contador; cuando ARRIBA y ABAJO son altos, el contador conserva su valor

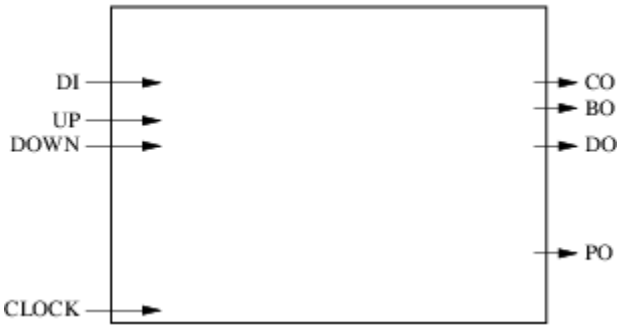
Nombre	Ancho de bits	Dirección	Descripción
correos	1	Salida	Paridad de señal de salida; alto cuando el valor actual del contador contiene un número par de 1

Cuando se cuenta más allá de su valor máximo o cuando se cuenta por debajo de su valor mínimo, el contador se ajusta:

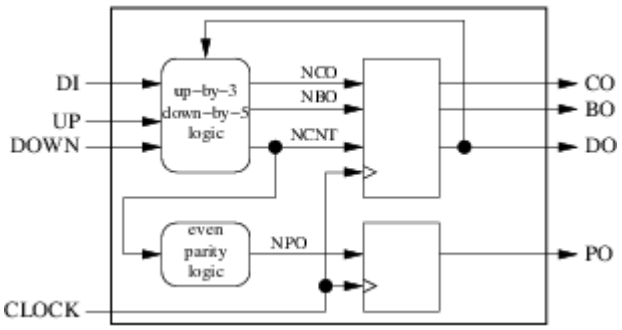
Contador de valor actual	ARRIBA ABAJO	Contador siguiente valor	Siguiente CO	Siguiente BO	Siguiente PO
X	00	DI	0	0	paridad (DI)
X	11	X	0	0	paridad (x)
0 ≤ x ≤ 508	10	x + 3	0	0	paridad (x + 3)
509	10	0	1	0	1
510	10	1	1	0	0
511	10	2	1	0	0
5 ≤ x ≤ 511	01	x-5	0	0	paridad (x – 5)
4	01	511	0	1	0
3	01	510	0	1	1
2	01	509	0	1	1
1	01	508	0	1	0
0	01	507	0	1	1

Diagrama de bloques

Basándonos en estas especificaciones podemos comenzar a diseñar un diagrama de bloques. Primero representemos la interfaz:

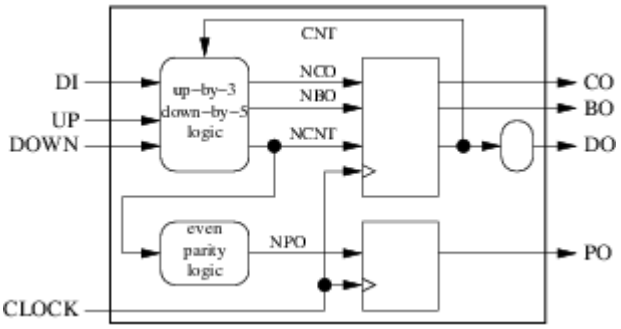


Nuestro circuito tiene 4 entradas (incluido el reloj) y 4 salidas. El siguiente paso consiste en decidir cuántos registros y bloques combinatorios usaremos y cuáles serán sus funciones. Para este ejemplo simple, dedicaremos un bloque combinatorio al cálculo del siguiente valor del contador, la ejecución y el préstamo. Se usará otro bloque combinatorio para calcular el siguiente valor de la paridad hacia afuera. Los valores actuales del contador, la ejecución y el préstamo se almacenarán en un registro, mientras que el valor actual de la paridad se almacenará en un registro separado. El resultado se muestra en la siguiente figura:



La comprobación del cumplimiento del diagrama de bloques con nuestras 10 reglas de diseño se realiza rápidamente:

1. Nuestra interfaz externa está representada adecuadamente por el gran rectángulo circundante.
2. Nuestros 2 bloques combinatorios (redondos) y nuestros 2 registros (cuadrados) están claramente separados.
3. Usamos solo registros activados por flanco ascendente.
4. Utilizamos un solo reloj.
5. Tenemos 4 flechas internas (señales), 4 flechas de entrada (puertos de entrada) y 4 flechas de salida (puertos de salida).
6. Ninguna de nuestras flechas tiene varios orígenes. Tres tienen varios destinos (`clock` , `ncnt` y `do`).
7. Ninguna de nuestras 4 flechas de entrada es una salida de nuestros bloques internos.
8. Tres de nuestras flechas de salida tienen exactamente un origen y un destino. Pero `do` con 2 destinos: el exterior y uno de nuestros bloques combinatorios. Esto infringe la regla número 8 y debe solucionarse insertando un nuevo bloque combinatorio si queremos cumplir con las versiones VHDL anteriores a 2008:



9. Tenemos ahora exactamente 5 señales internas (`cnt` , `nco` , `nbo` , `ncnt` y `npo`).
10. Solo hay un ciclo en el diagrama, formado por `cnt` y `ncnt` . Hay un bloque cuadrado en el ciclo.

Codificación en versiones VHDL anteriores a 2008

Traducir nuestro diagrama de bloques en VHDL es sencillo. El valor actual del contador varía de 0 a 511, por lo que utilizaremos una señal de `bit_vector` 9 bits para representarlo. La única sutileza proviene de la necesidad de realizar operaciones a nivel de bits (como calcular la paridad) y aritméticas en los mismos datos. El estándar `numeric_bit` paquete de biblioteca `ieee` resuelve esto: se declara un `unsigned` tipo con exactamente la misma declaración que `bit_vector` y sobrecarga los operadores aritméticos de tal manera que ellos toman cualquier mezcla de `unsigned` y números enteros. Para calcular la ejecución y el préstamo usaremos un valor temporal `unsigned` 10 bits.

Las declaraciones de la biblioteca y la entidad:

```
library ieee;
use ieee.numeric_bit.all;

entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;
```

El esqueleto de la arquitectura es:

```
architecture arc1 of cooley is
  signal cnt:  unsigned(8 downto 0);
  signal ncnt: unsigned(8 downto 0);
  signal nco:  bit;
  signal nbo:  bit;
  signal npo:  bit;
```

```
begin
    <...processes...>
end architecture arc1;
```

Cada uno de nuestros 5 bloques está modelado como un proceso. Los procesos síncronos correspondientes a nuestros dos registros son muy fáciles de codificar. Simplemente utilizamos el patrón propuesto en el ejemplo de codificación . El registro que almacena la bandera de paridad de salida, por ejemplo, está codificado:

```
  poreg: process(clock)
  begin
    if rising_edge(clock) then
      po <= npo;
    end if;
  end process poreg;
```

y el otro registro que almacena `co` , `bo` y `cnt` :

```
  cobocntreg: process(clock)
  begin
    if rising_edge(clock) then
      co  <= nco;
      bo  <= nbo;
      cnt <= ncnt;
    end if;
  end process cobocntreg;
```

El proceso combinatorio de cambio de nombre también es muy simple:

```
  rename: process(cnt)
  begin
    do <= (others => '0');
    do <= bit_vector(cnt);
  end process rename;
```

El cálculo de paridad puede usar una variable y un bucle simple:

```
  parity: process(ncnt)
    variable tmp: bit;
  begin
    tmp := '0';
    npo <= '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    npo <= not tmp;
  end process parity;
```

El último proceso combinatorio es el más complejo de todos, pero aplicar estrictamente el método de traducción propuesto también lo hace fácil:

```
  u3d5: process(up, down, di, cnt)
    variable tmp: unsigned(9 downto 0);
```

```
  begin
    tmp  := (others => '0');
    nco  <= '0';
    nbo  <= '0';
    ncnt <= (others => '0');
    if up = '0' and down = '0' then
      ncnt <= unsigned(di);
    elsif up = '1' and down = '1' then
      ncnt <= cnt;
    elsif up = '1' and down = '0' then
      tmp   := ('0' & cnt) + 3;
      ncnt  <= tmp(8 downto 0);
      nco   <= tmp(9);
    elsif up = '0' and down = '1' then
      tmp   := ('0' & cnt) - 5;
      ncnt  <= tmp(8 downto 0);
      nbo   <= tmp(9);
    end if;
  end process u3d5;
```

Tenga en cuenta que los dos procesos síncronos también podrían fusionarse y que uno de nuestros procesos combinatorios se puede simplificar en una simple asignación de señal concurrente. El código completo, con la biblioteca y las declaraciones de paquetes, y con las simplificaciones propuestas es el siguiente:

```
library ieee;
use ieee.numeric_bit.all;

entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;

architecture arc2 of cooley is
  signal cnt:  unsigned(8 downto 0);
  signal ncnt: unsigned(8 downto 0);
  signal nco:  bit;
  signal nbo:  bit;
  signal npo:  bit;
begin
  reg: process(clock)
  begin
    if rising_edge(clock) then
      co  <= nco;
      bo  <= nbo;
      po  <= npo;
      cnt <= ncnt;
    end if;
  end process reg;

  do <= bit_vector(cnt);
```

```
  parity: process(ncnt)
    variable tmp: bit;
  begin
    tmp := '0';
    npo <= '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    npo <= not tmp;
  end process parity;

  u3d5: process(up, down, di, cnt)
    variable tmp: unsigned(9 downto 0);
  begin
    tmp  := (others => '0');
    nco  <= '0';
    nbo  <= '0';
    ncnt <= (others => '0');
    if up = '0' and down = '0' then
      ncnt <= unsigned(di);
    elsif up = '1' and down = '1' then
      ncnt <= cnt;
    elsif up = '1' and down = '0' then
      tmp   := ('0' & cnt) + 3;
      ncnt  <= tmp(8 downto 0);
      nco   <= tmp(9);
    elsif up = '0' and down = '1' then
      tmp   := ('0' & cnt) - 5;
      ncnt  <= tmp(8 downto 0);
      nbo   <= tmp(9);
    end if;
  end process u3d5;
end architecture arc2;
```

Yendo un poco más lejos.

El método propuesto es simple y seguro, pero se basa en varias restricciones que se pueden relajar.

Salta el dibujo del diagrama de bloques

Los diseñadores experimentados pueden omitir el dibujo de un diagrama de bloques para diseños simples. Pero aún piensan primero en el hardware. Dibujan en su cabeza en lugar de en una hoja de papel, pero de alguna manera continúan dibujando.

Utilizar reinicios asíncronos.

Hay circunstancias en las que los reinicios asíncronos (o conjuntos) pueden mejorar la calidad de un diseño. El método propuesto admite solo restablecimientos sincrónicos (es decir, los restablecimientos que se tienen en cuenta en los flancos ascendentes del reloj):

```
process(clock)
begin
  if rising_edge(clock) then
    if reset = '1' then
      o <= reset_value_for_o;
    else
      o <= i;
    end if;
  end if;
end process;
```

La versión con reinicio asíncrono modifica nuestra plantilla agregando la señal de reinicio en la lista de sensibilidad y otorgándole la más alta prioridad:

```
process(clock, reset)
begin
  if reset = '1' then
    o <= reset_value_for_o;
  elsif rising_edge(clock) then
    o <= i;
  end if;
end process;
```

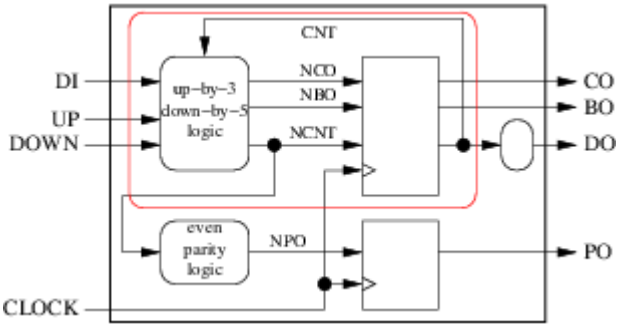
Fusionar varios procesos simples.

Ya lo usamos en la versión final de nuestro ejemplo. La fusión de varios procesos síncronos, si todos tienen el mismo reloj, es trivial. La fusión de varios procesos combinatorios en uno también es trivial y es solo una simple reorganización del diagrama de bloques.

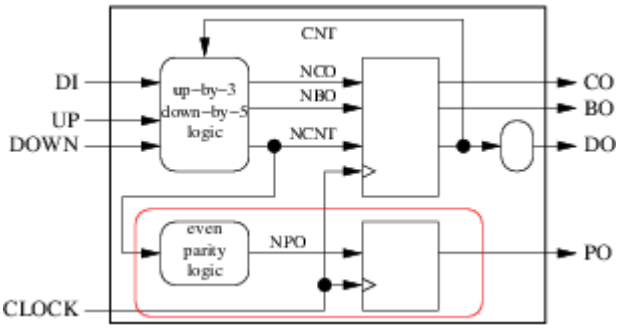
También podemos fusionar algunos procesos combinatorios con procesos síncronos. Pero para hacer esto debemos volver a nuestro diagrama de bloques y agregar una regla undécima:

11. Agrupe varios bloques redondos y al menos un bloque cuadrado dibujando un recinto alrededor de ellos. También encierra las flechas que pueden ser. No permita que una flecha cruce el límite del envolvente si no sale o va desde / hacia fuera del envolvente. Una vez hecho esto, mire todas las flechas de salida del gabinete. Si cualquiera de ellos proviene de un bloque redondo del gabinete o es también una entrada del gabinete, no podemos combinar estos procesos en un proceso síncrono. Si no podemos.

En nuestro ejemplo de contador, por ejemplo, no podríamos agrupar los dos procesos en el gabinete rojo de la siguiente figura:



Porque `ncnt` es una salida del gabinete y su origen es un bloque redondo (combinatorio). Pero podríamos agrupar:



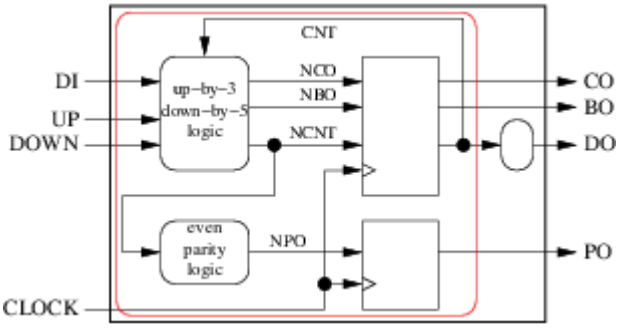
La señal interna `npo` se volvería inútil y el proceso resultante sería:

```
poreg: process(clock)
  variable tmp: bit;
begin
  if rising_edge(clock) then
    tmp := '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    po <= not tmp;
  end if;
end process poreg;
```

que también podría fusionarse con el otro proceso síncrono:

```
reg: process(clock)
  variable tmp: bit;
begin
  if rising_edge(clock) then
    co  <= nco;
    bo  <= nbo;
    cnt <= ncnt;
    tmp := '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    po <= not tmp;
  end if;
end process reg;
```

La agrupación podría incluso ser:



Conduciendo a la arquitectura mucho más simple:

```
architecture arc5 of cooley is
  signal cnt: unsigned(8 downto 0);
begin
  process(clock)
    variable ncnt: unsigned(9 downto 0);
    variable tmp:  bit;
  begin
    if rising_edge(clock) then
      ncnt := '0' & cnt;
      co   <= '0';
      bo   <= '0';
      if up = '0' and down = '0' then
        ncnt := unsigned('0' & di);
      elsif up = '1' and down = '0' then
        ncnt := ncnt + 3;
        co   <= ncnt(9);
      elsif up = '0' and down = '1' then
        ncnt := ncnt - 5;
        bo   <= ncnt(9);
      end if;
      tmp := '0';
      for i in 0 to 8 loop
        tmp := tmp xor ncnt(i);
      end loop;
      po  <= not tmp;
      cnt <= ncnt(8 downto 0);
    end if;
  end process;

  do <= bit_vector(cnt);
end architecture arc5;
```

con dos procesos (la asignación de señal concurrente de `do` es una abreviatura para el proceso equivalente). La solución con un solo proceso se deja como ejercicio. Cuidado, plantea preguntas interesantes y sutiles.

Yendo aún más lejos

Los pestillos activados por nivel, la caída de los bordes del reloj, los relojes múltiples (y los resincronizadores entre dominios de reloj), los controladores múltiples para la misma señal, etc. no son malos. A veces son útiles. Pero aprender a usarlos y cómo evitar las dificultades asociadas va mucho más allá de esta breve introducción al diseño de hardware digital con VHDL.

Codificación en VHDL 2008

VHDL 2008 introdujo varias modificaciones que podemos usar para simplificar aún más nuestro código. En este ejemplo podemos beneficiarnos de 2 modificaciones:

- Los puertos de salida se pueden leer, ya no necesitamos la señal `cnt` ,
- El operador `xor` unario se puede usar para calcular la paridad.

El código VHDL 2008 podría ser:

```
library ieee;
use ieee.numeric_bit.all;

entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;

architecture arc6 of cooley is
begin
  process(clock)
    variable ncnt: unsigned(9 downto 0);
  begin
    if rising_edge(clock) then
      ncnt := unsigned('0' & do);
      co   <= '0';
      bo   <= '0';
      if up = '0' and down = '0' then
        ncnt := unsigned('0' & di);
      elsif up = '1' and down = '0' then
        ncnt := ncnt + 3;
        co   <= ncnt(9);
      elsif up = '0' and down = '1' then
        ncnt := ncnt - 5;
        bo   <= ncnt(9);
      end if;
      po <= not (xor ncnt(8 downto 0));
      do <= bit_vector(ncnt(8 downto 0));
    end if;
  end process;
end architecture arc6;
```

Lea Diseño de hardware digital utilizando VHDL en pocas palabras en línea: https://riptutorial.com/es/vhdl/topic/5525/diseno-de-hardware-digital-utilizando-vhdl-en-pocas-palabras

Capítulo 6: Espere

Sintaxis

- espere [en SIGNAL1 [, SIGNAL2 [...]]] [hasta CONDITION] [para TIMEOUT];
- Espere; - eterna espera
- espera en s1, s2; - Espere hasta que las señales s1 o s2 (o ambas) cambien
- esperar hasta que s1 = 15; - Espere hasta que la señal s1 cambie y su nuevo valor sea 15
- esperar hasta que s1 = 15 por 10 ns; - Espere hasta que la señal s1 cambie y su nuevo valor sea 15 para un máximo de 10 ns

Examples

Espera eterna

La forma más simple de declaración de `wait` es simplemente:

```
wait;
```

Cada vez que un proceso ejecuta esto se suspende para siempre. El programador de simulación nunca lo reanudará de nuevo. Ejemplo:

```
signal end_of_simulation: boolean := false;
...
process
begin
  clock <= '0';
  wait for 500 ps;
  clock <= '1';
  wait for 500 ps;
  if end_of_simulation then
    wait;
  end if;
end process;
```

Listas de sensibilidad y frases de espera.

Un proceso con una lista de sensibilidad tampoco puede contener sentencias de espera. Es equivalente al mismo proceso, sin una lista de sensibilidad y con una última declaración más que es:

```
wait on <sensitivity_list>;
```

Ejemplo:

```
  process(clock, reset)
  begin
```

```
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process;
```

es equivalente a:

```
process
begin
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
  wait on clock, reset;
end process;
```

VHDL2008 introdujo `all` palabras clave en las listas de sensibilidad. Es equivalente a *todas las señales que se leen en algún lugar del proceso* . Es especialmente útil evitar listas de sensibilidad incompletas cuando se diseñan procesos combinatorios para la síntesis. Ejemplo de lista de sensibilidad incompleta:

```
process(a, b)
begin
  if ci = '0' then
    s  <= a xor b;
    co <= a and b;
  else
    s  <= a xnor b;
    co <= a or b;
  end if;
end process;
```

La señal `ci` no es parte de la lista de sensibilidad y esto es muy probable que sea un error de codificación que llevará a desajustes de simulación antes y después de la síntesis. El código correcto es:

```
process(a, b, ci)
begin
  if ci = '0' then
    s  <= a xor b;
    co <= a and b;
  else
    s  <= a xnor b;
    co <= a or b;
  end if;
end process;
```

En VHDL2008, la palabra clave `all` simplifica esto y reduce el riesgo:

```
process(all)
begin
```

```
    if ci = '0' then
      s  <= a xor b;
      co <= a and b;
    else
      s  <= a xnor b;
      co <= a or b;
    end if;
  end process;
```

Esperar hasta que la condicion

Es posible omitir las cláusulas `on <sensitivity_list>` y `for <timeout>` , como en:

```
    wait until CONDITION;
```

que es equivalente a:

```
    wait on LIST until CONDITION;
```

donde `LIST` es la lista de todas las **señales** que aparecen en `CONDITION` . También es equivalente a:

```
    loop
      wait on LIST;
      exit when CONDITION;
    end loop;
```

Una consecuencia importante es que si la `CONDITION` no contiene señales, entonces:

```
    wait until CONDITION;
```

es equivalente a:

```
    wait;
```

Un ejemplo clásico de esto es el famoso:

```
    wait until now = 1 sec;
```

eso no hace lo que uno podría pensar: como `now` es una función, no una señal, la ejecución de esta declaración suspende el proceso para siempre.

Espera una duración específica

utilizando solo la cláusula `for <timeout>` , es posible obtener una espera incondicional que dure por una duración específica. Esto no es sintetizable (ningún hardware real puede realizar este comportamiento de manera tan simple), pero se usa frecuentemente para programar eventos y generar relojes dentro de un banco de pruebas.

Este ejemplo genera un reloj de ciclo de trabajo de 100 MHz, 50% en el banco de pruebas de simulación para conducir la unidad bajo prueba:

```
constant period : time := 10 ns;
...
process
begin
   loop
      clk <= '0';
      wait for period/2;
      clk <= '1';
      wait for period/2;
   end loop;
end process;
```

Este ejemplo demuestra cómo se podría usar una espera de duración literal para secuenciar el proceso de análisis / estímulo de prueba:

```
process
begin
   rst <= '1';
   wait for 50 ns;
   wait until rising_edge(clk); --deassert reset synchronously
   rst <= '0';
   uut_input <= test_constant;
   wait for 100 us; --allow time for the uut to process the input
   if uut_output /= expected_output_constant then
      assert false report "failed test" severity error;
   else
      assert false report "passed first stage" severity note;
      uut_process_stage_2 <= '1';
   end if;
   ...
   wait;
end process;
```

Lea Espere en línea: https://riptutorial.com/es/vhdl/topic/6449/espere

Capítulo 7: Funciones de resolución, tipos no resueltos y resueltos.

Introducción

Los tipos de VHDL pueden ser *resueltos* o *resueltos* . El tipo de `bit` declarado por el paquete `std.standard` , por ejemplo, no se resuelve `ieee.std_logic_1164` se resuelve el tipo `std_logic` declarado por el paquete `ieee.std_logic_1164` .

Una señal cuyo tipo no está resuelto no puede ser activada (asignada) por más de un proceso VHDL, mientras que una señal cuyo tipo se resuelve puede.

Observaciones

El uso de los tipos resueltos debe reservarse a situaciones en las que la intención es realmente modelar un cable de hardware (o conjunto de cables) impulsado por más de un circuito de hardware. Un caso típico donde se necesita es el bus de datos bidireccional de una memoria: cuando se escribe la memoria, el dispositivo de escritura es el que impulsa el bus, mientras que cuando se lee la memoria es la memoria que impulsa el bus.

El uso de tipos resueltos en otras situaciones, aunque es una práctica frecuente, es una mala idea porque suprime los errores de compilación muy útiles cuando se crean accidentalmente situaciones de unidades múltiples no deseadas.

El paquete `ieee.numeric_std` declara los tipos de vectores `signed` y `unsigned` y sobrecarga los operadores aritméticos en ellos. Estos tipos se utilizan con frecuencia cuando se necesitan operaciones aritméticas y de bits en los mismos datos. Los tipos `signed` y `unsigned` se resuelven. Antes de VHDL2008, el uso de `ieee.numeric_std` y sus tipos implicaba, por lo tanto, que las situaciones accidentales de unidades múltiples no generarían errores de compilación. VHDL2008 agrega nuevas declaraciones de tipo a `ieee.numeric_std` : `unresolved_signed` y `unresolved_unsigned` (alias `u_signed` y `u_unsigned`). Estos nuevos tipos deberían ser preferidos en todos los casos donde no se deseen múltiples situaciones de unidad.

Examples

Dos procesos conducen la misma señal de tipo `bit`

Las siguientes unidades modelo VHDL señal `s` de dos procesos diferentes. Como el tipo de `s` es `bit` , un tipo sin resolver, esto no está permitido.

```
-- File md.vhd
entity md is
end entity md;
```

```
architecture arc of md is

  signal s: bit;

begin

  p1: process
  begin
    s <= '0';
    wait;
  end process p1;

  p2: process
  begin
    s <= '0';
    wait;
  end process p2;

end architecture arc;
```

Compilar, elaborar y tratar de simular, por ejemplo con GHDL, genera un error:

```
ghdl -a md.vhd
ghdl -e md
./md
for signal: .md(arc).s
./md:error: several sources for unresolved signal
./md:error: error during elaboration
```

Tenga en cuenta que el error se genera incluso si, como en nuestro ejemplo, todos los conductores están de acuerdo con el valor de conducción.

Funciones de resolución

Una señal del tipo que se resuelve tiene una *función de resolución* asociada. Puede ser impulsado por más de un proceso VHDL. La función de resolución se llama para calcular el valor resultante cada vez que un controlador asigna un nuevo valor.

Una función de resolución es una función pura que toma un parámetro y devuelve un valor del tipo a resolver. El parámetro es una matriz unidimensional, sin restricciones, de elementos del tipo a resolver. Para el `bit` tipo, por ejemplo, el parámetro puede ser de tipo `bit_vector` . Durante la simulación, se llama a la función de resolución cuando es necesario para calcular el valor resultante para aplicarla a una señal de activación múltiple. Se pasa una matriz de todos los valores controlados por todas las fuentes y devuelve el valor resultante.

El siguiente código muestra la declaración de una función de resolución para el tipo de `bit` que se comporta como un cableado `and` . También muestra cómo declarar un subtipo resuelto de tipo `bit` y cómo se puede usar.

```
-- File md.vhd
entity md is
end entity md;
```

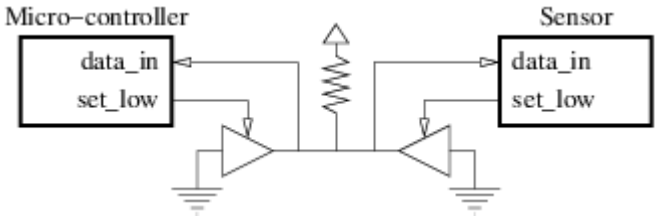
```
architecture arc of md is

  function and_resolve_bit(d: bit_vector) return bit is
    variable r: bit := '1';
  begin
    for i in d'range loop
      if d(i) = '0' then
        r := '0';
      end if;
    end loop;
    return r;
  end function and_resolve_bit;

  subtype res_bit is and_resolve_bit bit;

  signal s: res_bit;

begin

  p1: process
  begin
    s <= '0', '1' after 1 ns, '0' after 2 ns, '1' after 3 ns;
    wait;
  end process p1;

  p2: process
  begin
    s <= '0', '1' after 2 ns;
    wait;
  end process p2;

  p3: process(s)
  begin
    report bit'image(s); -- show value changes
  end process p3;

end architecture arc;
```

La compilación, elaboración y simulación, por ejemplo, con GHDL, no genera un error:

```
ghdl -a md.vhd
ghdl -e md
./md
md.vhd:39:5:@0ms:(report note): '0'
md.vhd:39:5:@3ns:(report note): '1'
```

Un protocolo de comunicación de un bit.

Algunos dispositivos de hardware muy simples y de bajo costo, como los sensores, utilizan un protocolo de comunicación de un bit. Una sola línea de datos bidireccional conecta el dispositivo a un tipo de microcontrolador. Es frecuentemente levantado por una resistencia de pull-up. Los dispositivos de comunicación conducen la línea baja durante un tiempo predefinido para enviar una información al otro. La siguiente figura ilustra esto:



Este ejemplo muestra cómo modelar esto utilizando el tipo resuelto `ieee.std_logic_1164.std_logic`
.

```
-- File md.vhd
library ieee;
use ieee.std_logic_1164.all;

entity one_bit_protocol is
end entity one_bit_protocol;

architecture arc of one_bit_protocol is

  component uc is
    port(
      data_in: in  std_ulogic;
      set_low: out std_ulogic
    );
  end component uc;

  component sensor is
    port(
      data_in: in  std_ulogic;
      set_low: out std_ulogic
    );
  end component sensor;

  signal data:           std_logic;  -- The bi-directional data line
  signal set_low_uc:     std_ulogic;
  signal set_low_sensor: std_ulogic;

begin

  -- Micro-controller
  uc0: uc port map(
    data_in => data,
    set_low => set_low_uc
  );

  -- Sensor
  sensor0: sensor port map(
    data_in => data,
    set_low => set_low_sensor
  );

  data <= 'H'; -- Pull-up resistor

  -- Micro-controller 3-states buffer
  data <= '0' when set_low_uc = '1' else 'Z';

  -- Sensor 3-states buffer
  data <= '0' when set_low_sensor = '1' else 'Z';

end architecture arc;
```

Lea Funciones de resolución, tipos no resueltos y resueltos. en línea:
https://riptutorial.com/es/vhdl/topic/9534/funciones-de-resolucion--tipos-no-resueltos-y-resueltos-

Capítulo 8: Identificadores

Examples

Identificadores basicos

Los identificadores básicos constan de letras, guiones bajos y dígitos y deben comenzar con una letra. No son sensibles a mayúsculas y minúsculas. Las palabras reservadas del idioma no pueden ser identificadores básicos. Ejemplos de identificadores básicos de VHDL válidos:

```
A_myId90
a_MYID90
abcDEf100_1
ABCdef100_1
```

Los dos primeros son equivalentes y los dos últimos también son equivalentes (insensibilidad a los casos).

Ejemplos de identificadores básicos inválidos:

```
_not_reset   -- start with underscore
85MHz_clock  -- start with digit
LooP         -- reserved word of the language
```

Identificadores extendidos

Los identificadores extendidos VHDL están delimitados por barras diagonales inversas (`\`) y pueden contener letras, guiones bajos, dígitos, espacios y otros caracteres especiales (consulte el Manual de consulta de idiomas para obtener una definición completa de los caracteres especiales). La secuencia de caracteres entre barras invertidas puede ser palabras reservadas del lenguaje VHDL. Las barras invertidas se pueden incluir en los identificadores extendidos duplicándolos (`\\`). Los identificadores extendidos distinguen entre mayúsculas y minúsculas. Ejemplos de identificadores extendidos (todos diferentes):

```
\if\
\If\
\My Identifier\
\An \\ Identifier \\ With \\ Backslashes\
\&#@[]:.*\
\$£$%{}\
```

Lea Identificadores en línea: https://riptutorial.com/es/vhdl/topic/9540/identificadores

Capítulo 9: Literales

Introducción

Esto tiene como especificar constantes, llamadas literales en VHDL.

Examples

Literales numericos

```
   16#A8# -- hex
   2#100# -- binary
   2#1000_1001_1111_0000 -- long number, adding (optional) _ (one or more) for readability
   1234 -- decimal
```

Literal enumerado

```
type state_t is (START, READING, WRITING); -- user-defined enumerated type
```

Lea Literales en línea: https://riptutorial.com/es/vhdl/topic/9344/literales

Capítulo 10: Recuerdos

Introducción

Esto cubre memorias de puerto único y de puerto dual.

Sintaxis

- Tipo de memoria para ancho y profundidad constante.

```
type MEMORY_TYPE is array (0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);
```

Tipo de memoria para profundidad variable y ancho constante.

```
type MEMORY_TYPE is array (natural range <>) of std_logic_vector(WIDTH-1 downto 0);
```

Examples

Registro de turnos

Un registro de desplazamiento de longitud genérica. Con serial in y serial fuera.

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

entity SHIFT_REG is
    generic(
        LENGTH: natural := 8
    );
    port(
        SHIFT_EN : in  std_logic;
        SO       : out std_logic;
        SI       : in  std_logic;
        clk      : in  std_logic;
        rst      : in  std_logic
    );
end entity SHIFT_REG;

architecture Behavioral of SHIFT_REG is
    signal reg : std_logic_vector(LENGTH-1 downto 0) := (others => '0');
begin
    main_process : process(clk) is
    begin
        if rising_edge(clk) then
            if rst = '1' then
                reg <= (others => '0');
            else
                if SHIFT_EN = '1' then
                    --Shift
```

```
                    reg <= reg(LENGTH-2 downto 0) & SI;
                else
                    reg <= reg;
                end if;
            end if;
        end if;
    end process main_process;

    SO <= reg(LENGTH-1);
end architecture Behavioral;
```

Para paralelo hacia fuera,

```
--In port
DOUT: out std_logic_vector(LENGTH-1 downto 0);
-----------------------------------------------
--In architecture
DOUT <= REG;
```

Registro de cambios con control de dirección, carga paralela, salida en paralelo. (Usando Variable en lugar de señal)

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

entity SHIFT_REG_UNIVERSAL is
    generic(
        LENGTH : integer := 8
    );
    port(
        DIN  : in  std_logic_vector(LENGTH - 1 downto 0);
        DOUT : out std_logic_vector(LENGTH - 1 downto 0);
        MODE : in  std_logic_vector(1 downto 0);
        SI   : in  std_logic;
        clk  : in  std_logic;
        rst  : in  std_logic
    );
end entity SHIFT_REG_UNIVERSAL;

architecture RTL of SHIFT_REG_UNIVERSAL is
begin
    main : process(clk, rst) is
        variable reg : std_logic_vector(LENGTH - 1 downto 0) := (others => '0');
    begin
        if rst = '1' then
            reg := (others => '0');
        elsif rising_edge(clk) then
            case MODE is
                when "00" =>
                    -- Hold Value
                    reg := reg;
                when "01" =>
                    -- Shift Right
                    reg := SI & reg(LENGTH - 1 downto 1);
                when "10" =>
                    -- Shift Left
                    reg := reg(LENGTH - 2 downto 0) & SI;
```

```
                when "11" =>
                    -- Parallel Load
                    reg := DIN;
                when others =>
                    null;
            end case;
        end if;
        DOUT <= reg;
    end process main;

end architecture RTL;
```

ROM

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

entity ROM is
    port(
        address : in  std_logic_vector(3 downto 0);
        dout    : out std_logic_vector(3 downto 0)
    );
end entity ROM;

architecture RTL of ROM is
    type MEMORY_16_4 is array (0 to 15) of std_logic_vector(3 downto 0);
    constant ROM_16_4 : MEMORY_16_4 := (
        x"0",
        x"1",
        x"2",
        x"3",
        x"4",
        x"5",
        x"6",
        x"7",
        x"8",
        x"9",
        x"a",
        x"b",
        x"c",
        x"d",
        x"e",
        x"f"
    );
begin
    main : process(address)
    begin
        dout <= ROM_16_4(to_integer(unsigned(address)));
    end process main;

end architecture RTL;
```

LIFO

Última en primera salida (pila) de memoria

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

entity LIFO is
    generic(
        WIDTH : natural := 8;
        DEPTH : natural := 128
    );
    port(
        I_DATA  : in  std_logic_vector(WIDTH - 1 downto 0); --Input Data Line
        O_DATA  : out std_logic_vector(WIDTH - 1 downto 0); --Output Data Line
        I_RD_WR : in  std_logic; --Input RD/~WR signal. 1 for READ, 0 for Write
        O_FULL  : out std_logic; --Output Full signal. 1 when memory is full.
        O_EMPTY : out std_logic; --Output Empty signal. 1 when memory is empty.
        clk     : in  std_logic;
        rst     : in  std_logic
    );
end entity LIFO;

architecture RTL of LIFO is
    -- Helper Function to convert Boolean to Std_logic
    function to_std_logic(B : boolean) return std_logic is
    begin
        if B = false then
            return '0';
        else
            return '1';
        end if;
    end function to_std_logic;

    type memory_type is array (0 to DEPTH - 1) of std_logic_vector(WIDTH - 1 downto 0);
    signal memory : memory_type;
begin
    main : process(clk, rst) is
        variable stack_pointer : integer range 0 to DEPTH := 0;
        variable EMPTY, FULL   : boolean                  := false;
    begin
        --Async Reset
        if rst = '1' then
            memory   <= (others => (others => '0'));
            EMPTY := true;
            FULL  := false;

            stack_pointer := 0;
        elsif rising_edge(clk) then
            if I_RD_WR = '1' then
                -- READ
                if not EMPTY then
                    O_DATA        <= memory(stack_pointer);
                    stack_pointer := stack_pointer - 1;
                end if;
            else
                if stack_pointer < 16 then
                    stack_pointer          := stack_pointer + 1;
                    memory(stack_pointer - 1) <= I_DATA;
                end if;
            end if;

            -- Check for Empty
            if stack_pointer = 0 then
```

```
                EMPTY := true;
            else
                EMPTY := false;
            end if;

            -- Check for Full
            if stack_pointer = DEPTH then
                FULL := true;
            else
                FULL := false;
            end if;
        end if;
        O_FULL  <= to_std_logic(FULL);
        O_EMPTY <= to_std_logic(EMPTY);
    end process main;

end architecture RTL;
```

Lea Recuerdos en línea: https://riptutorial.com/es/vhdl/topic/9521/recuerdos

Capítulo 11: Recursividad

Introducción

La recursividad es un método de programación donde los subprogramas se llaman a sí mismos. Es muy conveniente resolver algunos tipos de problemas de una manera elegante y genérica. VHDL soporta la recursividad. La mayoría de los sintetizadores lógicos también lo soportan. En algunos casos, el hardware inferido es incluso mejor (más rápido, del mismo tamaño) que con la descripción equivalente basada en bucles.

Examples

Cálculo del peso de Hamming de un vector.

```
-- loop-based version
function hw_loop(v: std_logic_vector) return natural is
  variable h: natural;
begin
  h := 0;
  for i in v'range loop
    if v(i) = '1' then
      h := h + 1;
    end if;
  end loop;
  return h;
end function hw_loop;

-- recursive version
function hw_tree(v: std_logic_vector) return natural is
  constant size: natural := v'length;
  constant vv: std_logic_vector(size - 1 downto 0) := v;
  variable h: natural;
begin
  h := 0;
  if size = 1 and vv(0) = '1' then
    h := 1;
  elsif size > 1 then
    h := hw_tree(vv(size - 1 downto size / 2)) + hw_tree(vv(size / 2 - 1 downto 0));
  end if;
  return h;
end function hw_tree;
```

Lea Recursividad en línea: https://riptutorial.com/es/vhdl/topic/10775/recursividad

Capítulo 12: Tipos protegidos

Observaciones

Antes de VHDL 1993, dos procesos concurrentes solo podían comunicarse con señales. Gracias a la semántica de simulación del lenguaje que actualiza las señales solo entre los pasos de simulación, el resultado de una simulación fue determinista: no dependía del orden elegido por el programador de simulación para ejecutar los procesos.

[De hecho, esto no es 100% cierto. Los procesos también pueden comunicarse utilizando la entrada / salida de archivos. Pero si un diseñador estaba comprometiendo el determinismo mediante el uso de archivos, realmente no podría ser un error.]

El lenguaje era así seguro. Excepto a propósito, fue casi imposible diseñar modelos VHDL no deterministas.

VHDL 1993 introdujo variables compartidas y el diseño de modelos VHDL no deterministas se hizo muy fácil.

VHDL 2000 introdujo los tipos protegidos y la restricción de que las variables compartidas deben ser de tipo protegido.

En VHDL, los tipos protegidos son lo que más se asemeja al concepto de objetos en lenguajes orientados a objetos (OO). Implementan la encapsulación de estructuras de datos y sus métodos. También garantizan el acceso exclusivo y atómico a sus miembros de datos. Esto no evita completamente el no determinismo pero, al menos, agrega exclusividad y atomicidad a las variables compartidas.

Los tipos protegidos son muy útiles cuando se diseñan modelos VHDL de alto nivel diseñados solo para simulación. Tienen varias propiedades muy buenas de idiomas OO. Su uso frecuente hace que el código sea más legible, mantenible y reutilizable.

Notas:

- Algunas cadenas de herramientas de simulación, de forma predeterminada, solo emiten advertencias cuando una variable compartida no es de un tipo protegido.
- Algunas herramientas de síntesis no admiten tipos protegidos.
- Algunas herramientas de síntesis tienen un soporte limitado de variables compartidas.
- Se podría pensar que las variables compartidas no se pueden usar para modelar hardware y deben reservarse para instrumentación de código sin efectos secundarios. Pero los patrones VHDL recomendados por varios proveedores de EDA para modelar el plano de memoria de memorias de acceso aleatorio (RAM) de múltiples puertos utilizan variables compartidas. Entonces, sí, las variables compartidas pueden ser sintetizables en ciertas circunstancias.

Examples

Un generador pseudoaleatorio.

Los generadores peudoaleatorios suelen ser útiles para diseñar entornos de simulación. El siguiente paquete VHDL muestra cómo usar tipos protegidos para diseñar un generador pseudoaleatorio de `boolean` , `bit` y `bit_vector` . Se puede extender fácilmente para generar también `std_ulogic_vector` aleatorio, `signed` , `unsigned` . Extenderlo para generar enteros aleatorios con límites arbitrarios y una distribución uniforme es un poco más complicado pero factible.

La declaración del paquete

Un tipo protegido tiene una declaración donde se declaran todos los accesores públicos de subprogramas. Para nuestro generador aleatorio, haremos público un procedimiento de inicialización de inicialización y tres funciones impuras que devuelven un `boolean` aleatorio, `bit` o `bit_vector` . Tenga en cuenta que las funciones no pueden ser puras, ya que las diferentes llamadas de cualquiera de ellas, con los mismos parámetros, pueden devolver valores diferentes.

```
-- file rnd_pkg.vhd
package rnd_pkg is
    type rnd_generator is protected
        procedure init(seed: bit_vector);
        impure function get_boolean return boolean;
        impure function get_bit return bit;
        impure function get_bit_vector(size: positive) return bit_vector;
    end protected rnd_generator;
end package rnd_pkg;
```

El cuerpo del paquete

El cuerpo de tipo protegido define las estructuras de datos internas (miembros) y los cuerpos del subprograma. Nuestro generador aleatorio se basa en un Registro de Cambio de Retroalimentación Lineal (LFSR) de 128 bits con cuatro pulsaciones. La variable de `state` almacena el estado actual del LFSR. Un procedimiento de `throw` privado cambia el LFSR cada vez que se usa el generador.

```
-- file rnd_pkg.vhd
package body rnd_pkg is
    type rnd_generator is protected body
        constant len: positive := 128;
        constant default_seed: bit_vector(1 to len) := X"8bf052e898d987c7c31fc71c1fc063bc";
        type tap_array is array(natural range <>) of positive range 1 to len;
        constant taps: tap_array(0 to 3) := (128, 126, 101, 99);

        variable state: bit_vector(1 to len) := default_seed;

        procedure throw(n: positive := 1) is
            variable tmp: bit;
        begin
            for i in 1 to n loop
                tmp := '1';
```

```
                for j in taps'range loop
                    tmp := tmp xnor state(taps(j));
                end loop;
                state := tmp & state(1 to len - 1);
            end loop;
        end procedure throw;

        procedure init(seed: bit_vector) is
            constant n:   natural            := seed'length;
            constant tmp: bit_vector(1 to n) := seed;
            constant m:   natural            := minimum(n, len);
        begin
            state         := (others => '0');
            state(1 to m) := tmp(1 to m);
        end procedure init;

        impure function get_boolean return boolean is
            constant res: boolean := state(len) = '1';
        begin
            throw;
            return res;
        end function get_boolean;

        impure function get_bit return bit is
            constant res: bit := state(len);
        begin
            throw;
            return res;
        end function get_bit;

        impure function get_bit_vector(size: positive) return bit_vector is
            variable res: bit_vector(1 to size);
        begin
            if size <= len then
                res := state(len + 1 - size to len);
                throw(size);
            else
                res(1 to len) := state;
                throw(len);
                res(len + 1 to size) := get_bit_vector(size - len);
            end if;
            return res;
        end function get_bit_vector;
    end protected body rnd_generator;
end package body rnd_pkg;
```

El generador aleatorio se puede usar en un estilo OO como en:

```
-- file rnd_sim.vhd
use std.env.all;
use std.textio.all;
use work.rnd_pkg.all;

entity rnd_sim is
end entity rnd_sim;

architecture sim of rnd_sim is
    shared variable rnd: rnd_generator;
begin
    process
```

```
        variable l: line;
    begin
        rnd.init(X"fe39_3d9f_24bb_5bdc_a7d0_2572_cbff_0117");
        for i in 1 to 10 loop
            write(l, rnd.get_boolean);
            write(l, HT);
            write(l, rnd.get_bit);
            write(l, HT);
            write(l, rnd.get_bit_vector(10));
            writeline(output, l);
        end loop;
        finish;
    end process;
end architecture sim;
```

```
$ mkdir gh_work
$ ghdl -a --std=08 --workdir=gh_work rnd_pkg.vhd rnd_sim.vhd
$ ghdl -r --std=08 --workdir=gh_work rnd_sim
TRUE    1    0001000101
FALSE   0    1111111100
TRUE    1    0010110010
TRUE    1    0010010101
FALSE   0    0111110100
FALSE   1    1101110010
TRUE    1    1011010110
TRUE    1    0010010010
TRUE    1    1101100111
TRUE    1    0011100100
simulation finished @0ms
```

Lea Tipos protegidos en línea: https://riptutorial.com/es/vhdl/topic/6362/tipos-protegidos

Creditos

S. No	Capítulos	Contributors
1	Empezando con vhdl	Community, DavideM, Florian, Jon Klapel, Josh, Renaud Pacalet
2	Análisis de tiempo estático: ¿qué significa cuando un diseño falla en el tiempo?	QuantumRipple
3	Comentarios	Renaud Pacalet
4	D-Flip-Flops (DFF) y cierres	Brian Carlton, Renaud Pacalet
5	Diseño de hardware digital utilizando VHDL en pocas palabras	Renaud Pacalet
6	Espere	Brian Carlton, QuantumRipple, Renaud Pacalet
7	Funciones de resolución, tipos no resueltos y resueltos.	Renaud Pacalet
8	Identificadores	Renaud Pacalet
9	Literales	Brian Carlton, QuantumRipple
10	Recuerdos	Brian Carlton, Parth Parikh, QuantumRipple, Renaud Pacalet
11	Recursividad	Renaud Pacalet
12	Tipos protegidos	Renaud Pacalet