Revista de Ingeniería y Ciencias Aplicadas. (RevICAp)

Página web: https://revistas.uncu.edu.ar/ojs3/index.php/revicap

Croscompilación Python de VHDL para promover el aprendizaje de Hardware Reconfigurables

Jaime Alberto Parra-Plaza[1*]

1. Grupo de Investigación en Bionanoelectrónica, Universidad del Valle, Cali, Colombia.

*E-mail: jaimealberto.parra@gmail.com

PALABRAS CLAVES

Croscompilación
Hardware reconfigurable
Lenguaje de descripción de hardware
Python

RESUMEN

Los dispositivos de hardware reconfigurable han revolucionado la forma en que se diseñan sistemas electrónicos a la medida y han permitido lograr soluciones que aprovechan al máximo recursos limitados como la batería o el espacio. Sin embargo, lograr la experticia en estos dispositivos es un reto tanto para la industria como para la academia. Si bien actualmente existen herramientas que proveen la posibilidad de diseños en alto nivel, la eficiencia lograda en la solución final con este método no es la más conveniente, de allí que sea aún necesario trabajar en buena medida con diseños en bajo nivel, particularmente con lenguajes de descripción de hardware, los cuales han mostrado ser particularmente difíciles de aprender para muchos estudiantes en ingeniería electrónica. En este artículo se presentan los resultados de una investigación cuyo objetivo fue la creación de una herramienta de croscompilación que permite a los alumnos describir sus sistemas hardware en el lenguaje de alto nivel Python. La herramienta genera una versión VHDL apropiada para sintetizar un procesador a la medida con uso óptimo de recursos. Los resultados indican mejoras en la tasa de aprendizaje en términos de interés, motivación y asimilación, siguiendo el modelo de estilos de aprendizaje de Kolb.

Python cross-compilation of VHDL to foster learning of Reconfigurable Hardware

KEYWORDS

Cross-compilation
Reconfigurable hardware
Hardware description language
Python

ABSTRACT

Reconfigurable hardware devices have revolutionized the way custom electronic systems are designed, enabling solutions that maximize the use of limited resources, such as the battery life of portable devices or the space available in functional blocks that must operate in a macro system with significant area constraints. However, achieving expertise in the management of these devices is a challenge for both industry and academia. While tools currently exist that provide high-level design capabilities, the efficiency achieved in the final solution with this method is not optimal. Therefore, it is still necessary to work largely with low-level designs, particularly with hardware description languages, which have proven particularly difficult to learn for many electronic engineering students. This paper presents the results of a research project aimed at creating a cross-compilation tool that allows students to describe their hardware systems in a high-level language such as Python. The tool generates a VHDL version suitable for synthesizing a custom processor with optimal resource utilization. The results indicate an improvement in the learning rate given in terms of interest, motivation and assimilation, following Kolb's learning styles model.

1. Introducción

La electrónica constituye uno de los pilares de la sociedad actual. Gracias a ella, ha sido posible automatizar un sinfín de procesos y lograr potencias de cómputo cada vez mayores con dispositivos que son consistentemente más pequeños y más rápidos. En el caso particular de la electrónica digital, el dispositivo estrella fue, y todavía lo es en gran medida, el microprocesador. Gracias a él, los ingenieros han podido crear sistemas cuyo poder de cómputo supera con creces a los grandes y monumentales equipos de cómputo que hace tan

solo unas pocas décadas ocupaban salas enteras y costaban millones de dólares (O'Regan, 2021).

Sin embargo, las prestaciones de los microprocesadores actuales, y por ende su costo, exceden por mucho las necesarias para gran cantidad de soluciones en donde, si bien se requiere cierto poder de cómputo, éste es sólo una fracción del disponible en un microprocesador convencional. Para cubrir ese segmento se han propuesto diferentes alternativas, tales como los microcontroladores o el hardware a la medida. No obstante, y dado que el cambio generacional entre una solución y la siguiente versión se mide actualmente en meses o incluso en semanas, cuando antes era en años, surgió hace algunas décadas un fuerte competidor en este segmento: los dispositivos de hardware reconfigurable (Blokdyk, 2018). El hardware reconfigurable constituye en principio el sueño hecho realidad de todo diseñador de sistemas digitales. En un dispositivo se tiene un arreglo de compuertas lógicas que inicialmente tienen una capacidad omnipotente de conectarse unas con otras en cualquier configuración arbitraria, de allí que también se conozca como FPGA por la sigla en inglés de arreglo de compuertas programable en campo, una de las estructuras reconfigurables más populares en este tipo de dispositivos (ver Figura 1).

Una FPGA está constituida por un conjunto de CLBs o bloques lógicos configurables, que son pequeños arreglos de circuitos digitales típicos, tales como un grupo de compuertas AND, OR y XOR, junto con algunos dispositivos de memoria como latches y flip-flops. Entre los CLB discurren, a manera de autopistas, líneas de conexión que pueden seleccionarse a voluntad para que conecten un CLB en particular con otro. Si bien este nivel de detalle es apropiado para el diseñador mismo de los FPGA, el ingeniero digital usa el dispositivo como una caja negra e interactúa con ella a través de una herramienta de configuración de alto nivel (Nicolescu y Mosterman, 2018).

De esta manera, el diseñador puede ensayar diferentes configuraciones circuitales para que cuando eventualmente logre una que satisfaga los requerimientos de diseño, decidir si procede a fabricarlo en una pastilla de silicio dedicada o, en el caso de que sólo se requieran algunas unidades, dejar que el propio diseño logrado en la FPGA sea a la vez el dispositivo de trabajo. Si bien la cantidad de reconfiguraciones que se puede hacer en la práctica no es infinita, sí son suficientes para que un diseñador avezado logre obtener resultados en un tiempo prudencial. Lograr la experticia en el diseño digital usando FPGA requiere el dominio de un conjunto variado de técnicas y herramientas y una en particular se muestra particularmente retadora para los estudiantes de ingeniería que entran al mundo del diseño de sistemas digitales. Es en este punto en donde la necesidad del presente proyecto entra en escena. El proyecto, denominado Py2HDL ofrece a los alumnos una herramienta tecnológica que permite abordar el aprendizaje del diseño de sistemas en hardware reconfigurable de una forma más afín a los postulados de las ciencias pedagógicas en cuanto a la forma en que el aprendizaje deviene en el ser humano. Py2HDL permite a los alumnos adquirir incrementalmente las destrezas necesarias para comprender y dominar el lenguaje VHDL partiendo de un lenguaje más accesible como es Python, junto con un conjunto de dinámicas basadas en el modelo de aprendizaje de Kolb.

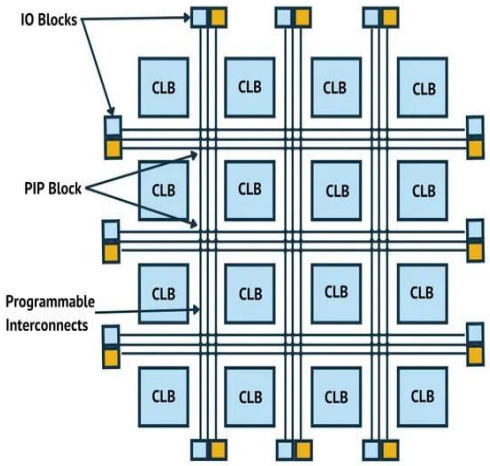

Figura 1. Estructura interna de un dispositivo de hardware reconfigurable.

2. Materiales y métodos

Entorno de desarrollo para FPGA

La complejidad de las FPGA actuales es tal que literalmente están constituidas por cientos de miles de compuertas lógicas, por lo que abordar un diseño a ese nivel es impensable. Es por ello que los fabricantes de las mismas ofrecen herramientas de cómputo muy elaboradas que se encargan de manejar el dispositivo a nivel de compuerta, al tiempo que ofrecen al diseñador una interfaz que le permite interactuar con el dispositivo a nivel de sistema (ver Figura 2).

Por otra parte, al igual que un microprocesador es útil en la medida en que se conectan periféricos a él, por ejemplo teclados, pantallas, interfaces de comunicación, etc., de la misma manera, en la

práctica, el diseño con dispositivos FPGA se hace empleando tarjetas de desarrollo, elaboradas ya sea por los mismos fabricantes o por terceros. Estos módulos de desarrollo contienen periféricos para que la comunicación con la FPGA sea más cómoda. Periféricos típicos son por ejemplo un conjunto de interruptores y pulsadores a manera de entrada, arreglos de LEDs o pantallas LCD a manera de salida y diversas interfaces de comunicación como ethernet, USB o bluetooth (ver Figura 3).

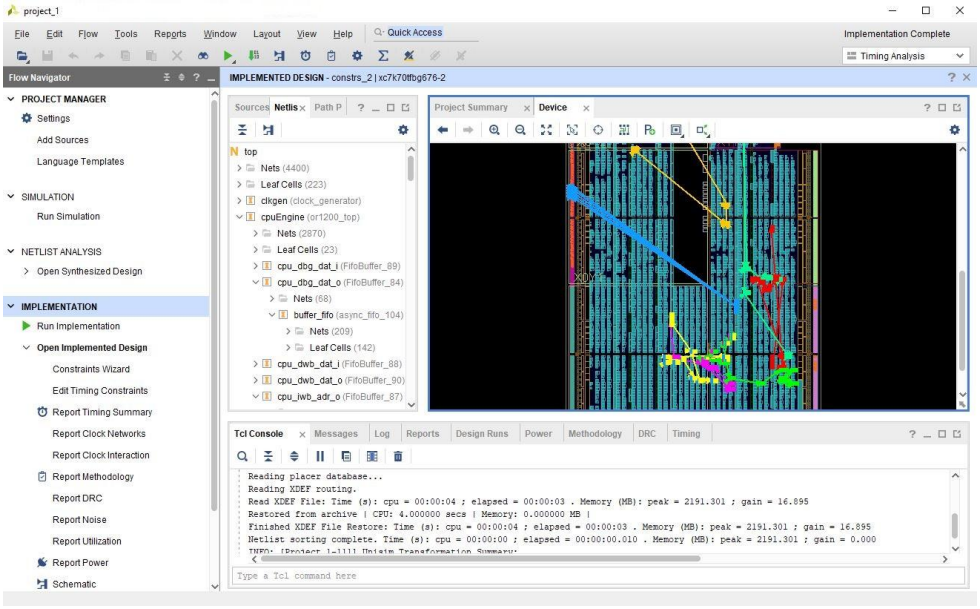

Figura 2. Interfaz típica de un entorno de desarrollo para hardware reconfigurable

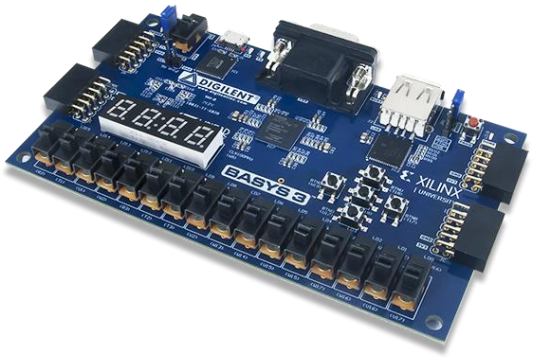

Figura 3. Tarjeta de desarrollo típica para interactuar con un dispositivo de hardware reconfigurable.

Lenguajes de descripción de hardware

Así como una FPGA es compleja en su estructura circuital, es tanto o más compleja cuando está en operación como un sistema lógico secuencial. Basta sólo con pensar en la interacción simultánea de sus miles de celdas lógicas para darse una idea de la magnitud comportamental que puede obtenerse. Dado el paralelismo extremo que aparece, emplear simples métodos de diseño como tablas de verdad o máquinas de estado es sólo posible para diseños muy simples. Para diseños medianos a grandes, se hace indispensable un nivel de abstracción mucho mayor. Es por ello que los fabricantes y los diseñadores digitales acogieron en un momento dado una opción que en principio se usó para la documentación de dispositivos tipo VLSI, pero que se ha popularizado y ha encontrado su nicho en el mundo del hardware reconfigurable: los lenguajes de descripción de hardware o HDL (Pedroni, 2016).

Un lenguaje de programación convencional describe las operaciones que realiza un microprocesador ya sea real o virtual de forma secuencial. Lenguajes típicos son C, C++, Java o Python. Por su parte, un HDL es un lenguaje que describe la forma en que un circuito digital está estructurado. Si bien su apariencia, en términos de sintaxis, puede llegar a hacerlo similar a un lenguaje de programación, su propósito y uso son bastante diferentes. Estas diferencias suelen ser pasadas por alto por los estudiantes que recién inician a usar el lenguaje e incluso por diseñadores con más experiencia. Esto se debe en gran medida al hecho de que nuestro cerebro, al menos en su parte consciente, comprende mucho más fácilmente los procesos que son secuenciales. Aquéllos que son concurrentes, por su propia definición, requieren ser analizados como si se tomara una foto o instantánea de un momento particular (Jasinski, 2016).

A pesar de que se han propuesto diversos lenguajes de descripción de hardware, hay dos que sobresalen

y son por mucho los más populares en el mundo del diseño digital: VHDL y Verilog. En el caso de Latinoamérica, la preferencia es por el primero de ellos y ese es el lenguaje que se empleó en el presente proyecto. Se procede ahora a analizar brevemente el HDL en sí. Para una introducción más detallada a los conceptos fundamentales del lenguaje, consultar (Parra-Plaza, 2020). VHDL es un lenguaje que se basa en la definición de bloques lógicos estructurales. Cada bloque está compuesto ya sea por descripciones explícitas de operaciones lógicas combinacionales o por descripciones en alto nivel de comportamientos secuenciales. El lenguaje es altamente jerárquico y permite anidar estructuras a cualquier nivel de profundidad que se desee. Las únicas operaciones nativas son las operaciones lógicas booleanas tradicionales. Los demás constructos se realizan por operaciones de decisión, mediante el clásico if, y por operaciones de iteración, al hacer uso explícito de la retroalimentación de variables, que en el caso de VHDL se designan más apropiadamente como señales, para indicar su naturaleza física. Esta mezcla, que en principio es muy conveniente para el diseñador experimentado, constituye, según los análisis que el autor ha realizado, una de las fuentes de mayor confusión en el momento en que el aprendiz intenta comprender apropiadamente el funcionamiento de un sistema descrito en VHDL (Parra-Plaza, 2015). Considere por caso el fragmento de código presentado en la Figura 4.

```
if pol='1' then          ①②③④
  a <= buf1;             ①  ③
else
  a <= not buf1;         ②  ④
end if;

if act='1' then          ①②③④
  res <= a or b;         ①     ④
else
  res <= a and b;        ②③
end if;
```

Figura 4. Fragmento de código en VHDL ilustrando los problemas de concurrencia.

En principio se observa que en términos de sintaxis este código es parecido a cualquier otro lenguaje convencional de programación. Si se analiza de esa forma, es decir como si la ejecución fuese secuencial, línea por línea, se tiene que pasaría por el primer if, a partir del cual la variable a toma algún valor y, cuando se ingresa al segundo if, ese valor de a se usaría para asignar un valor subsecuente a la variable res. Sin embargo, este no es el caso en el lenguaje VHDL. Dado que él no describe instrucciones a ser ejecutadas, sino que describe hardware, se debe analizar como si ambos if se están ejecutando en paralelo. De esta manera, las variables a y res, en tanto que señales, dada su naturaleza física, están cambiando simultáneamente. Esta forma de comprender el funcionamiento de un código en VHDL es bastante retadora y el diseñador debe hacer un ejercicio constante de recordar esta característica, en tanto la asimile y la haga parte de su experticia (Parra-Plaza, 2012).

Croscompilación software

El primer paso en el proceso de diseño de un sistema digital es lograr una descripción apropiada del mismo que satisfaga los diferentes requerimientos funcionales, comportamentales y físicos (Parra-Plaza, 2016). En el caso del presente proyecto, esta actividad culmina cuando se tiene un código Python que describe esa funcionalidad. Se eligió Python como lenguaje de descripción de alto nivel porque satisface la mayoría de características que el autor y su equipo definieron para tal propósito (Parra-Plaza, 2023), teniendo en la cuenta además que el presente proyecto articula con otros proyectos en un macroproyecto destinado a promover el aprendizaje significativo mediante herramientas tecnológicas, en el cual Python se emplea transversalmente (Parra-Plaza, 2018). Es claro que los fabricantes de FPGA son conscientes de la necesidad de disponer de lenguajes y herramientas de descripción de alto nivel, pero las que se ofrecen actualmente, tales como SystemC (Black et al., 2009) o Amaranth (Amaranth Project, 2023), están aún muy ligadas al hardware y requieren que el diseñador describa explícitamente estructuras como el reloj del sistema o el número de bits de cada variable. Esas características son indeseables a la luz de la filosofía que guía este proyecto y eso reafirma la decisión de emplear Python en su forma nativa, sin requerir ningún tipo de modificación ni en su sintaxis ni en su procesamiento (Parra-Plaza, 2019).

En informática se habla de dos conceptos base con respecto a los lenguajes de programación: lenguajes de alto nivel y lenguajes de bajo nivel o de máquina. Los primeros se emplean por los diseñadores humanos para concebir un código, llamado fuente, que representa la funcionalidad del sistema a diseñar, los segundos los usa el hardware en sí mismo para ejecutar el código de máquina y ejecutar las instrucciones que implementan el diseño especificado. La labor de traducir un código fuente a un código máquina se denomina compilación. En el presente proyecto se emplea un concepto distinto, pero relacionado con el mismo, la croscompilación,

la cual consiste en tomar un código fuente en un lenguaje de alto nivel y traducirlo a otro código fuente en otro lenguaje de alto nivel (Kleitz, 2011). Para el caso presente, el código fuente es Python y el código destino es VHDL. Considérese como ejemplo un diseño simple, el de un comparador, que realiza la comparación entre dos valores y genera por salida el mayor de ellos. El código fuente de Python se muestra en la Figura 5. Se basa en una función que toma dos parámetros, a y b, y determina mediante una comparación directa si se debe devolver el valor del primero o del segundo parámetro. Para mostrar el uso de variables locales, una de ellas se emplea para almacenar el valor a devolver. Igualmente, por fuera de la función, es permitido generar código que invoque a la función, capture su respuesta y la exhiba en pantalla. Este código externo a las funciones se usará más adelante por parte de la herramienta para procesos de validación y verificación funcional y comportamental.

```
1  # sample file for pyhdl: if simple
2
3  def comp(a, b):
4      temp = b
5      if a>b:
6          temp = a
7      return temp
8
9  x = comp(5, 4)
10 print x
11
```

Figura 5. Código fuente del comparador.

Tras la compilación cruzada, la herramienta, que en adelante se identificará como Py2HDL, genera el archivo comp.vhd, el cual contiene el código VHDL para la función Python comp. Este archivo se compone de varios bloques de descripción, así que se analizará parte por parte. En primer lugar se tiene la entidad, la cual describe la interfaz del sistema digital con su entorno, indicando, además de su nombre, sus señales de entrada y de salida (ver Figura 6). Obsérvese la presencia del genérico N el cual define la cantidad de bits que tendrán las distintas señales. Este valor es generado automáticamente por el croscompilador teniendo en la cuenta la capacidad en periféricos de la tarjeta de desarrollo destino. Cada tarjeta posee cierto número de interruptores, los cuales se suelen emplear para ingresar datos hacia la FPGA. Py2HDL reconoce esta información y la usa para calcular cuántos bits puede asignar a cada señal, de tal manera que cada señal quede representada y que se maximice la cantidad de bits a usar. Igualmente, se tiene la presencia de otras señales distintas a las señales originales de datos a y b. Estas señales (start, clock, reset, ready, result), conocidas como señales de control, son necesarias para el correcto funcionamiento del hardware y para sincronizar sus procesos en los tiempos adecuados. Como se indicó anteriormente, los lenguajes de alto nivel usados por los fabricantes exigen al diseñador manejar directamente estas señales. Py2HDL intencionalmente oculta estos detalles específicos de hardware al estudiante en sus primeros encuentros con las FPGA. Posteriormente estarán disponibles para que él los asuma, en la medida en que sea procedente, siguiendo los estilos de aprendizaje de Kolb (Kolb y Kolb, 2017).

La funcionalidad VHDL asociada a la función Python comp se describe en la arquitectura, cuyo encabezado, indicando los estados que tendrá el sistema secuencial, se muestra en la Figura 7. El nombre de los estados se genera basándose en el tipo de sentencias y en los números de línea presentes en el archivo fuente para facilitar la asignación entre ellos con fines de depuración o mejora. Por ejemplo, la asignación "temp = b", que ocurre en la línea 4 del código Python, se relaciona con el estado "assign4", la condición "if a>b", que ocurre en la línea 5, se realiza en el estado "if5", y así sucesivamente. Las variables locales se asignan a señales con el número adecuado de bits según la definición de los parámetros.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity comp is
6      generic(N : integer := 4);
7      port(start : in std_logic;
8           clock : in std_logic;
9           reset : in std_logic;
10          a : in signed(N-1 downto 0);
11          b : in signed(N-1 downto 0);
12          result : out signed(N-1 downto 0);
13          ready : out std_logic);
14 end comp;
15
```

Figura 6. Código VHDL de destino generado por Py2HDL para el comparador (entidad).

```
16 architecture arch of comp is
17     type state is (idle, assign4, if5, assign6, return7, finish);
18     signal currentState, nextState : state;
19     signal temp : signed(N-1 downto 0);
20
```

Figura 7. Código VHDL de destino generado por Py2HDL para el comparador (señales).

La implementación de la funcionalidad se realiza mediante un procesador hecho a la medida o custom, el cual está diseñado como una ruta de datos o datapath junto con una máquina de estados de tipo Moore que hará las veces de controlador. La ruta de datos se encarga de rastrear las asignaciones en el momento oportuno, teniendo en la cuenta los cambios en las variables del archivo fuente (véase la

Figura 8). La máquina de estados es sincrónica. Utiliza un reloj maestro y un reinicio síncrono para gestionar los cambios de estado. Py2HDL inserta automáticamente un estado inactivo inicial, llamado idle, en el que la máquina permanece hasta que sea activada por la señal de inicio (véase la Figura 9).

```
21 begin
22     -- datapath
23     process(clock, currentState)
24     begin
25         if (clock'event and clock = '1') then
26             if (currentState = assign4) then
27                 temp <= b;
28             end if;
29             if (currentState = assign6) then
30                 temp <= a;
31             end if;
32             if (currentState = return7) then
33                 result <= temp;
34             end if;
35         end if;
36     end process;
37
```

Figura 8. Código VHDL de destino generado por Py2HDL para el comparador (ruta de datos).

```
38     -- state register
39     process(clock, reset)
40     begin
41         if (reset = '1') then
42             currentState <= idle;
43         elsif (clock'event and clock = '1') then
44             currentState <= nextState;
45         end if;
46     end process;
47
```

Figura 9. Código VHDL de destino generado por Py2HDL para el comparador (registro de estado).

La evolución de la máquina de estados depende del cálculo del siguiente estado (véase la Figura 10). Para ello se tienen en la cuenta en la lista de sensibilidad todas las señales capaces de generar eventos. Se puede insertar un estado de finalización adicional si el diseñador desea que la máquina permanezca en este estado, para efectos de probar el comportamiento del hardware. El enfoque estándar es dejar que la máquina vuelva al estado inactivo para que esté disponible inmediatamente para un nuevo cálculo.

Cuando el valor a devolver es estable, el módulo informa a su entorno dicho evento para que otros módulos actúen de conformidad. Esto se hace activando una señal de bandera, "ready", creada automáticamente por Py2HDL (ver Figura 11).

```
48     -- next-state logic
49     process(currentState, start, temp)
50     begin
51         case currentState is
52         when idle =>
53             if (start = '1') then
54                 nextState <= assign4;
55             else
56                 nextState <= idle;
57             end if;
58         when assign4 =>
59                 nextState <= if5;
60         when assign6 =>
61                 nextState <= return7;
62         when return7 =>
63                 nextState <= finish;
64         when if5 =>
65             if (a > b) then
66                 nextState <= assign6;
67             else
68                 nextState <= return7;
69             end if;
70         when finish =>
71             nextState <= finish;
72         end case;
73     end process;
74
```

Figura 10. Código VHDL de destino generado por Py2HDL para el comparador (lógica del estado siguiente).

```
75     -- output logic
76     ready <= '1' when (currentState = finish) else '0';
77 end arch;
78
```

Figura 11. Código VHDL de destino generado por Py2HDL para el comparador (lógica de salida).

Simulación comportamental

Previo a implementar el diseño en hardware, conviene realizar una simulación del mismo. A diferencia de una solución software, en donde una simulación funcional en suficiente, una solución hardware requiere una simulación comportamental, en la cual se evalúe que el diseño no sólo realice la función asignada, sino también que la haga en los tiempos tanto absolutos como relativos necesarios. Un simulador muy apreciado para VHDL es ModelSim, el cual, aunque pasa por algunos cambios al haber sido adquirido por una nueva empresa, continúa siendo un referente en el mundo VHDL. El simulador requiere disponer de dos archivos: El primer archivo es el archivo de descripción y el segundo es un banco de pruebas o testbench que instruye al simulador con respecto a los valores que debe asignar a cada señal de entrada y a los tiempos en que debe hacerlo. Para una comprensión más detallada del proceso de simulación ver (Parra-Plaza, 2021).

Py2HDL genera un archivo TB que contiene el banco de pruebas para un diseño dado. El contenido de este archivo se genera a partir de las instrucciones Python que detecte por fuera de funciones, en particular llamados a las funciones e instrucciones print. El archivo compTB.vhd incluye las pruebas de alto nivel realizadas en la función comp. Crea señales para todos los puertos en comp.vhd, junto con constantes y señales que simulan valores y variables para las pruebas de alto nivel. Por ejemplo, probar x = comp(5, 4) requiere la creación de la señal x y de las constantes 5 y 4. Py2HDL ordena las constantes incrementalmente para facilitar su inspección. También añade valores para el periodo de la señal de reloj y asigna todas las señales a una instancia de la función (ver Figura 12).

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

entity compTB is
end compTB;

architecture tb_arch of compTB is
   constant T : time    := 20 ns; -- clk period = 50 MHz
   constant N : integer := 4;
   signal TB_start : std_logic;
   signal TB_clock : std_logic;
   signal TB_reset : std_logic;
   signal TB_a : signed(N-1 downto 0);
   signal TB_b : signed(N-1 downto 0);
   signal TB_result : signed(N-1 downto 0);
   signal TB_ready : std_logic;
   constant S4 : signed(N-1 downto 0) := to_signed(4, N);
   constant S5 : signed(N-1 downto 0) := to_signed(5, N);
   signal x : signed(N-1 downto 0);

begin
   -- instantiate the circuit under test
   uut: entity work.comp(arch)
      generic map(N=>N)
      port map(start=>TB_start, clock=>TB_clock, reset=>TB_reset, a=>TB_a, b=>TB_b, result=>TB_result, ready=>TB_ready);

```

Figura 12. Código VHDL de banco de pruebas generado por Py2HDL para el comparador (señales y componente).

Py2HDL genera una señal de reloj similar a la del hardware real y activa la señal de reset durante el primer periodo de reloj para un inicio predictivo limpio (ver Figura 13). Las pruebas siguen una secuencia de asignaciones que se asimilan a las pruebas de alto nivel: se dan valores a los argumentos, se genera la señal de inicio y se permite que se estabilice, se pasa el control al módulo y se programa el evento de retorno detectando el valor de la señal listo, el resultado se asigna a la señal de prueba y se deja estable durante varios ciclos de reloj para que esté dispone a otros módulos (ver Figura 14).

```
   -- clock generation
   process
   begin
      TB_clock <= '1';
      wait for T/2;
      TB_clock <= '0';
      wait for T/2;
   end process;

   -- reset
   TB_reset <= '1', '0' after T;

```

Figura 13. Código VHDL de banco de pruebas generado por Py2HDL para el comparador (reloj y reset).

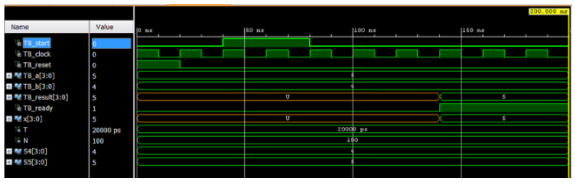

Figura 14. Resultado de la simulación para el comprador.

Síntesis hardware

Disponer sólo del archivo vhd que describa la funcionalidad del diseño no es suficiente para su implementación final. Para ello, la herramienta de desarrollo debe realizar un proceso denominado síntesis, el cual consiste en asignar recursos o bloques lógicos de la FPGA a cada constructo que logre identificar en la descripción dada en VHDL (Rushton, 2011). Para poder hacer eso es indispensable que la herramienta conozca cuál es la tarjeta de desarrollo sobre la cual se hará la implementación, de tal manera que pueda asociar los diferentes periféricos de entrada y de salida con las señales internas correspondientes. Para una comprensión más detallada del proceso de síntesis ver (Parra-Plaza, 2022).

Py2HDL presta su ayuda en la síntesis durante dos momentos clave. Primero, selecciona sólo las construcciones VHDL que tengan sentido en hardware. VHDL tiene múltiples propósitos, algunos de los cuales tienen más relación con procesos de simulación y de documentación que con implementación en hardware. Py2HDL elige un subconjunto de VHDL, denominado VHDL sintetizable, para que la herramienta de síntesis realice apropiadamente su trabajo. Segundo, crea un archivo de tipo XHC. Este archivo instruye a la herramienta de síntesis para que realice un mapeado entre los recursos externos a la FPGA y las señales internas de la misma.

Para el presente proyecto, la tarjeta de desarrollo empleada es una Nexys4, ampliamente utilizada en la academia por sus buenas prestaciones y bajo costo. El archivo comp.xhc es la asignación física para el caso del comparador. Py2HDL determina qué señales deben proporcionarse para uso externo y calcula la cantidad de bits disponibles para cada una en función del hardware de destino. Py2HDL asigna la señal de reloj al oscilador de cristal presente en la placa y le indica que genere una señal simétrica. También asigna las señales de reinicio (reset) e inicio (start) a los pulsadores, la señal de listo (ready) al LED más alto y la señal de resultado (result) al LED más bajo requerido (ver Figura 15).

```
# Mapping CLOCK to OSC
set_property LOC E3 [get_ports clock]
set_property IOSTANDARD LVCMOS33 [get_ports clock]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clock]

# Mapping RESET to BTND
set_property LOC V10 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
# Mapping START to BTNU
set_property LOC F15 [get_ports start]
set_property IOSTANDARD LVCMOS33 [get_ports start]
# Mapping ready to LED<15>
set_property LOC P2 [get_ports ready]
set_property IOSTANDARD LVCMOS33 [get_ports ready]

# Mapping result[0] to LED<0>
set_property LOC T8 [get_ports {result[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[0]}]
# Mapping result[1] to LED<1>
set_property LOC V9 [get_ports {result[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[1]}]
# Mapping result[2] to LED<2>
set_property LOC R8 [get_ports {result[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[2]}]
# Mapping result[3] to LED<3>
set_property LOC T6 [get_ports {result[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[3]}]

```

Figura 15. Código XHC de asignación física generado por PyHDL para el comparador (señales del sistema).

Con respecto a las señales de parámetros, Py2HDL calcula la cantidad de bits que cada una puede contener en función de las variables Python de alto nivel y de la cantidad de recursos disponibles en el hardware. Para ello inicia con las señales de salida, las cuales son mapeadas hacia el conjunto de LEDs, teniendo en la cuenta además que, según el tipo de operaciones que detecte en el código fuente, asignará un ancho de bits igual o duplicado al de las señales de entrada. Posteriormente realiza una labor similar con las señales de entrada. Esto puede conllevar a que se recalcule la cantidad de bits en las salidas si detecta que la asignación inicial impide una asignación completa de las entradas. La versión actual de Py2HDL privilegia los escenarios de aprendizaje más que su uso por diseñadores más experimentados, por lo cual el LED más significativo se ha empleado para señalizar la finalización en la ejecución del código. Esto con miras a detectar posibles bucles infinitos. Para el caso de la placa Nexys4, la asignación detallada de recursos hardware se muestra en la Figura 16.

```
# Mapping a to SW<0>
set_property LOC U9 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[0]}]
# Mapping a to SW<1>
set_property LOC U8 [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[1]}]
# Mapping a to SW<2>
set_property LOC R7 [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[2]}]
# Mapping a to SW<3>
set_property LOC R6 [get_ports {a[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[3]}]
# Mapping b to SW<4>
set_property LOC R5 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
# Mapping b to SW<5>
set_property LOC V7 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
# Mapping b to SW<6>
set_property LOC V6 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[2]}]
# Mapping b to SW<7>
set_property LOC V5 [get_ports {b[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[3]}]

```

Figura 16. Código XHC de asignación física generado por PyHDL para el comparador (señales del diseñador).

Una vez la herramienta sintetiza el diseño, crea un archivo en código objeto de configuración que puede ser descargado a la placa de hardware. Al momento en que esta descarga finaliza, la configuración ya permanece en la tarjeta hardware indefinidamente hasta que sea remplazada por alguna nueva configuración. En estas condiciones, la tarjeta ya actúa como un sistema independiente, y la ejecución del diseño ya es posible. La Figura 17 muestra el resultado tras pulsar la señal de inicio cuando los argumentos proporcionados son los valores 5 y 2 (binarios 0101 y 0010).

Como segunda prueba, se cambiaron los valores a 5 y 7 (binarios 0101 y 0111). El patrón de los LEDs muestra el cambio en el resultado. El LED más alto indica que el resultado es estable y que está disponible para su uso seguro (véase la Figura 18). Para mayor comodidad al momento de evaluar un diseño determinado, Py2HDL le permite al diseñador generar una secuencia de pruebas en lote. Si en el código fuente se detecta la presencia de varias

instrucciones print, Py2HDL generará automáticamente un bloque contador que actuará como un pequeño temporizador de tal manera que, una vez el resultado de la primera prueba esté disponible, se le dé tiempo al diseñador de inspeccionarlo visualmente y corroborar si es correcto o no y determinar si procede continuar las pruebas o regresar al proceso de diseño. En la versión actual de Py2HDL este valor de temporizado se fijó en 2 segundos, lo cual es suficiente para comprobar la mayoría de resultados en un primer vistazo.

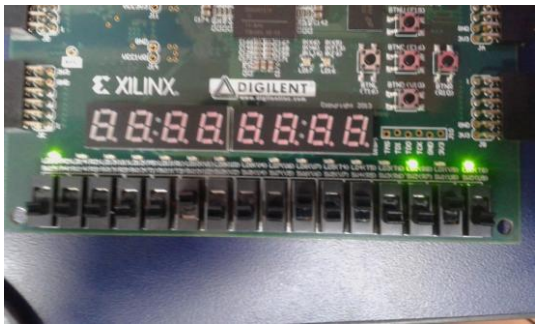

Figura 17. Ejecución de hardware para la prueba del comparador (argumentos: a=5, b=2).

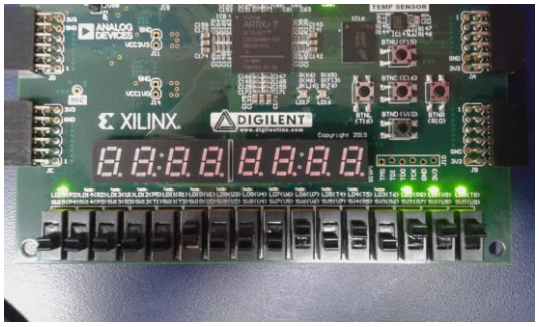

Figura 18. Ejecución de hardware para la prueba del comparador (argumentos: a=5, b=7).

Varias otras pruebas fueron realizadas con diseños de mediana y alta complejidad. Éstos incluyen comparadores de tres y más valores, divisor, calculador de raíz cuadrada, detector de clave de acceso y una versión del juego de picas y famas (Parra-Plaza, 2013). Las limitaciones están dadas por la propia naturaleza del dispositivo. Por ejemplo, las operaciones matemáticas actúan sobre operadores enteros y la cantidad de entradas y de salidas depende de las que puedan alojarse en el conjunto de pulsadores, interruptores y LEDs de que disponga la tarjeta. Más que verlos como limitantes, es un reconocimiento del tipo de aplicaciones para las cuales una FPGA es conveniente. Para soluciones que requieran cálculos en punto flotante, un microprocesador sería una mejor elección; y para soluciones que involucren también señales analógicas, un buen candidato podría ser un dispositivo del tipo PSoC (Van Ess, 2014).

La Figura 19 ilustra el resultado al ejecutar el código que extrae la raíz cuadrada de una variable num, para el caso en que num tiene un valor de 82 (binario 1010010), el cual se puede observar en la disposición de los interruptores. Al finalizar la ejecución, indicada por la activación del LED más significativo, el resultado se observa en los cuatro LEDs menos significativos, que contienen el valor binario 1001, correspondiente al valor decimal 9, el cual es la raíz entera más cercana a la raíz cuadrada de 82.

Figura 19. Ejecución de hardware para la prueba de raíz cuadrada (argumento: num=82).

3. Resultados y Discusión

Es posible para determinar el impacto que el empleo intencional de la herramienta de croscompilación podría tener en la mejor asimilación del contenido conceptual y en el desarrollo de habilidades de diseño en hardware reconfigurable y así ulteriormente en el aprendizaje, se realizaron diferentes pruebas a grupos de alumnos. En todos los casos, cada alumno interactuó con Py2HDL en diversas circunstancias, obedeciendo a los estilos de aprendizaje de Kolb (2017). En particular, el modelo de Kolb describe cuatro estilos de aprendizaje, denominados Divergente, Asimilador, Convergente y Acomodador, los cuales a su vez derivan de un Ciclo de Aprendizaje Experiencial, cuyas etapas son la Experiencia Concreta, la Observación Reflexiva, la Conceptualización Abstracta y la Experimentación Activa. Para el caso del presente proyecto, se hizo énfasis en el uso de la herramienta para las dos primeras etapas.

En esencia, Kolb adhiere al concepto constructivista (Doyle y Zakrajsek, 2013) en cuanto a que el aprendizaje deviene en una construcción, en esencia en la formación de redes neuronales específicas en el cerebro del aprendiz, para soportar la

comprensión y elaboración del concepto o habilidad que se está aprendiendo. Como tal, el constructivismo privilegia una primera aproximación práctica al objeto de estudio, antes que un discurso conceptual, en contravía con muchas tendencias largamente usadas en el sector educativo. Fieles a ese concepto, en esta investigación se ofrece al estudiante una aproximación al diseño en hardware reconfigurable partiendo de una aplicación que le permite experimentar de primera mano el diseño hardware en VHDL (objetivo), partiendo de un conocimiento ya adquirido como es el lenguaje Python (punto de partida). De esta manera, se busca despertar las asociaciones pertinentes que favorezcan una inserción más natural a los conceptos más abstractos del hardware, siguiendo los postulados de una de las vertientes más conocidas del constructivismo, el aprendizaje significativo (Ausubel et al, 1978).

Las actividades se desarrollaron simultáneamente por los alumnos, para lo cual se establecieron tres grupos, cada uno de 25 alumnos: A) grupo control: alumnos que no usaron Py2HDL, B) alumnos que usaron Py2HDL en su versión estándar, C) alumnos que usaron Py2HDL en su versión minimalista. Ésta es una versión que deliberamente oculta al alumno elementos de la plataforma que no sean pertinentes para su momento. En primera instancia, la interfaz que se ofrece es absolutamente elemental, al estilo del conocido buscador Google, una pantalla en donde solo hay una casilla para escribir código Python y un botón de ejecución, distinto a la versión estándar de la herramienta que es más afín a un IDE (entorno de desarrollo) convencional.

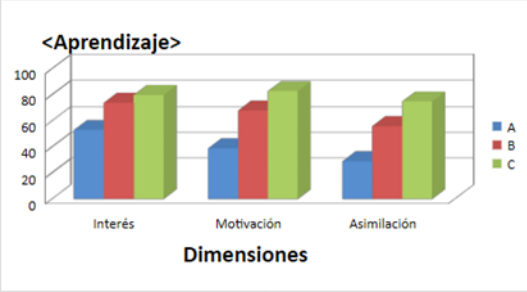

Figura 20. Métricas de aprendizaje para los primeros grupos considerados.

La Figura 20 muestra los resultados de las medidas de aprendizaje realizadas, los valores son el promedio por cada grupo y están dados en porcentajes, donde el 100% es el valor perfecto de la categoría. Se tomaron tres aspectos: interés, motivación y asimilación (Parra-Plaza, 2017). El interés se midió como el tiempo promedio que permanecía el alumno en una sesión de trabajo, la motivación es el número de ejercicios que realizó de los disponibles en la sesión y la asimilación se determinó como la cantidad de respuestas correctas que obtuvo el alumno en una prueba de suficiencia realizada un mes después de haber estudiado el tema.

Se observa que el empleo de la herramienta como vehículo de interacción para desarrollar habilidades de diseño genera mejorías con respecto al enfoque convencional de dar la teoría y luego ir a sesiones de laboratorio, contrastando los valores para los grupos A y B. A su vez, se observa cómo el empleo de la interfaz minimalista evidenció mejorías ulteriores, tal como indica los mayores valores del grupo C con respecto al grupo B. La diferencia más notable se da en la asimilación, en donde la menor distracción y stress que produce esta interfaz hace que la retención y aplicación ulterior de conocimientos más que se duplique con respecto a la forma convencional de instrucción, incluso siendo ésta mediada tecnológicamente.

Con miras a establecer en qué medida el uso de las etapas en el modelo de Kolb puede beneficiar aun más el proceso, se incluyó un cuarto grupo (D). En este grupo se realizó la etapa de Observación Reflexiva. Para ello, Py2HDL fue dotado de un módulo adicional que se beneficia de una aplicación de inteligencia artificial para establecer un diálogo con el aprendiz en cuanto a la experiencia vivida en las sesiones con el entorno de diseño. De nuevo, este diálogo se buscó que fuera también enfocado a un minimalismo, para lo cual la aplicación IA se alimentó con material adecuado para un diálogo simple pero reflexivo al respecto (Parra-Plaza, 2019).

Con esta novedad se obtuvo la distribución que se indica en la Figura 21. Se observa que los resultados para el grupo D superan a todos los otros grupos. Este resultado sugiere que la mediación tecnológica aunada a paradigmas o modelos de aprendizaje coherentes con esta mediación son una alianza que beneficia tanto los procesos mismos de enseñanza como el desempeño a lograr por los alumnos.

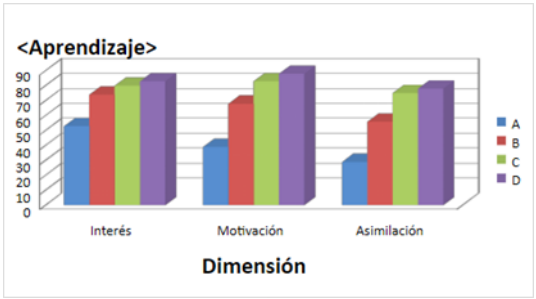

Figura 21. Métricas de aprendizaje para todos los grupos considerados.

4. Conclusiones

Es posible mejorar el desarrollo de habilidades cognitivas realizando una exposición incremental en la complejidad de la información y de los procesos a comprender por parte de los estudiantes. De esta manera, se da tiempo al cerebro de que construya las redes neuronales correspondientes de acuerdo con qué tan significativo sea uno u otro aspecto del contenido presentado y de las actividades realizadas.

La construcción de conocimiento que logre perdurar en el tiempo puede facilitarse al mediar los procesos educativos con soluciones tecnológicas creativas que se basen en paradigmas pedagógicos que tengan en cuenta las actividades experienciales. En la medida en que las actividades didácticas estén guiadas por un fundamento neurobiológico tanto del aprendizaje como de la enseñanza, es posible, de manera intencionada, influir positivamente en el proceso de desarrollo de habilidades perdurables por parte de los estudiantes.

El empleo de Py2HDL como herramienta de intervención en el proceso educativo manifestó ofrecer resultados de mejora en las distintas dimensiones del aprendizaje que se consideraron, como son interés, motivación y asimilación, medidas tanto inmediatamente como posteriormente en el tiempo, indicando una demostración en la práctica de los principios del constructivismo en general y del aprendizaje significativo en particular.

La inserción del modelo de aprendizaje de Kolb, en combinación con herramientas tecnológicas que faciliten su realización, se mostró como una posibilidad que incrementa la construcción de conocimiento, el desarrollo de habilidades y la retención ulterior de información y conceptos clave en una determinada disciplina.

La comprensión de la concurrencia al emplear VHDL para el desarrollo de sistemas digitales se propicia mediante herramientas tecnológicas que permitan al alumno observar un paralelismo en la ejecución entre dos sistemas: el que está aprendiendo y otro que le sea más familiar. La complejidad asociada con el paralelismo intenso que subyace a todos los procesos que ocurren en un sistema digital, aunada a la complejidad comportamental cuando presenta retroalimentación, es más abordable si se dispone de un referente que esté más cercano a la cotidianidad y a la representación de esquemas y propuestas, como es el caso de Python.

Disponer de herramientas de croscompilación permite establecer diferentes escenarios de aprendizaje en donde el protagonismo del aprendiz vaya gradualmente haciéndose más importante al pasar de conceptos básicos a intermedios y complejos. El uso intencional de herramientas de tecnología educativa permite modular la intensidad y la transición entre estas etapas.

5. Referencias

Amaranth Project (2023). Disponible en: amaranth-lang.org/docs/amaranth/latest/guide.html

Ausubel, D. Novak, J. y Hanesian, H. (1978). Educational Psychology: A Cognitive View. Holt, Rinehart & Winston.

Based Learning Systems.

Black, D. C., Donovan, J., Bunton, B. y Keist, A. (2009). SystemC: From the ground up. Springer.

Blokdyk, G. (2018). Hardware-Reconfigurable Devices. 5STARCooks.

Doyle, T. y Zakrajsek, T. (2013). The New Science of Learning: How to Learn in Harmony With Your Brain. Stylus Publishing.

Jasinski, R. (2016). Effective Coding with VHDL: Principles and Best Practice. MIT Press.

Kleitz, W. (2011). Digital Electronics: A Practical Approach with VHDL. Pearson.

Kolb, A. Y. y Kolb, D. A. (2017). The Experiential Educator: Principles and Practices of Experiential Learning. Experience

Nicolescu, G. y Mosterman, P. J. (2018). Model-Based Design for Embedded Systems. CRC Press.

O'Regan, G. A. (2021). Brief History of Computing. Springer.

Parra-Plaza, J. A. (2012). Concurrent programming: towards an optimal computation. EIISI.

Parra-Plaza, J. A. (2013). VHDL implementation of the Cows and Bulls game. Technical report. Cali (Colombia). Pontificia Universidad Javeriana.

Parra-Plaza, J. A. (2015). High-level Synthesis Through a Cross-compiler from Pure Python to Hardware Description Languages. WCAS.

Parra-Plaza, J. A. (2016). Custom processors design using Python-based high level synthesis. Instituto Antioqueño de Investigación.

Parra-Plaza, J. A. (2018). Computación adaptativa para mediar tecnológicamente en la enseñanza para el aprendizaje. Instituto Antioqueño de Investigación.

Parra-Plaza, J. A. (2019). Citoaprendizagem: Computação bioinspirada adaptativa focada na aprendizagem significativa. CISCI.

Parra-Plaza, J. A. (2020). Introducción a VHDL. Disponible en: www.youtube.com/watch?v=ZCX20VK5GmE

Parra-Plaza, J. A. (2021). Simulación VHDL con ModelSim. Disponible en: www.youtube.com/watch?v=KTISNd4NCWo

Parra-Plaza, J. A. (2022). Síntesis VHDL con Vivado. Disponible en: www.youtube.com/watch?v=Q2MCcWfLNhM

Parra-Plaza, J. A. (2023). PyHDL: Cross-compiler from pure Python to Hardware Description Languages. Technical Report. Institución Universitaria Antonio José Camacho.

Parra-Plaza, J.A. (2017). Propiciando el aprendizaje significativo en entornos interactivos mediante la inserción de moduladores neurogenéticos. Compdes.

Pedroni, V. A. (2020). Circuit Design with VHDL. MIT Press.

Rushton, A. (2011). VHDL for Logic Synthesis. Wiley.

Van Ess, D. (2014). Learn Digital Design with PSoC, a bit at a time. Pearson.

Zalar, P., Gostinčar, C., de Hoog, G. S., Uršič, V., Sudhadham, M., & Gunde-Cimerman, N. (2008). Redefinition of Aureobasidium pullulans and its varieties. Studies in Mycology, 61, 21–38. https://doi.org/10.3114/sim.2008.61.02