Sistemas Empotrados

2025-2026

Manual de programación de Verilog y API

Universidad de Alicante

José Vicente Berná Martínez

jvberna@ua.es

Contenido

Instalación en LINUX

Instalación de **APIO** en cualquier distribución de Linux (suponemos que **Python2** y **pip** está instalado):

Vamos a instalar usando el gestor de paquetes pip de Python, el entorno APIO (https://apiodoc.readthedocs.io/en/stable) en un **subdirectorio local** de nuestro usuario.

```
$ pip install -U apio
```

Solución al Error externally-managed-environment

El error externally-managed-environment que estás viendo en tu sistema Ubuntu es una protección introducida por Debian/Ubuntu para evitar que los paquetes de Python instalados a través de pip interfieran con los paquetes de Python gestionados por el propio sistema operativo (instalados a través de apt). Hay dos formas de solucionarlo.

Opción 1

Si realmente necesitas instalar el paquete globalmente (o no quieres configurar un entorno virtual), puedes anular esta protección usando el flag --break-system-packages. Aunque este flag te permite instalar el paquete, es importante saber que podría causar problemas si el paquete apio o sus dependencias sobrescriben o entran en conflicto con librerías del sistema. Úsalo con precaución.

```
pip install -U apio --break-system-packages
```

Opción 2

La forma más segura y recomendada de instalar paquetes de Python como apio que no son del sistema es dentro de un entorno virtual (venv). Esto aísla la instalación de apio y sus dependencias del resto del sistema operativo, eliminando el riesgo de conflictos.

1.- Crea el Entorno Virtual: Navega hasta la carpeta de tu proyecto de Verilog y ejecuta:

```
python3 -m venv venv_apio
```

2.- Activa el Entorno: Para empezar a usar el entorno, actívalo con:

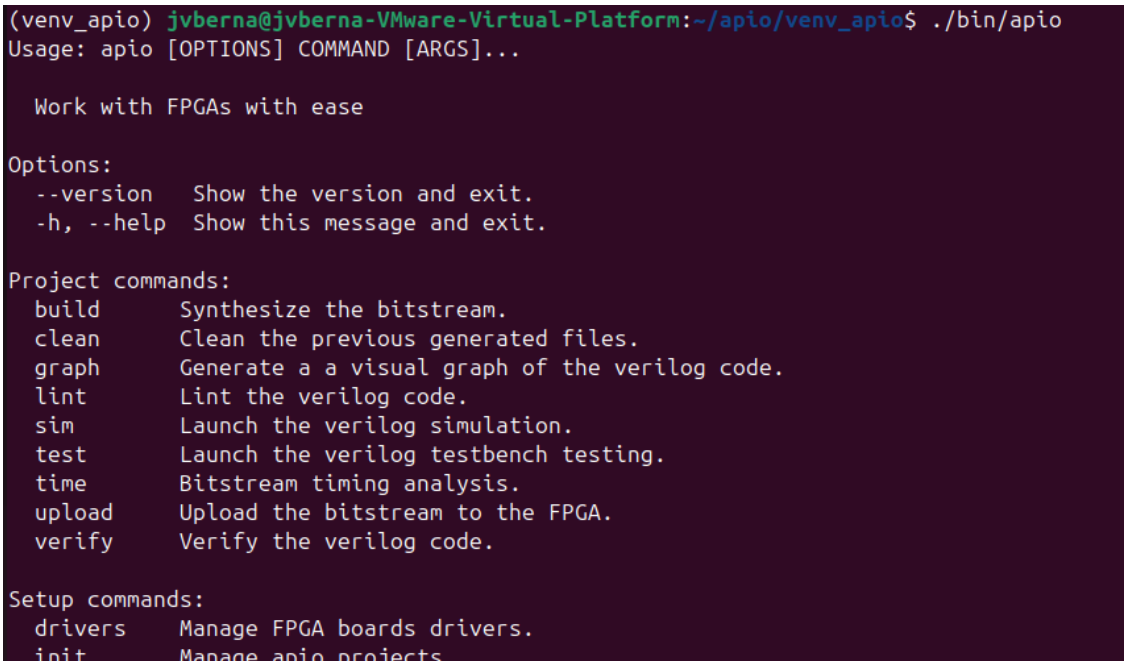
```
source venv_apio/bin/activate
```

Verás que el nombre del entorno (venv_apio) aparece al inicio de tu prompt de terminal.

3.- Instala apio: Ahora, dentro del entorno virtual, la instalación funcionará sin el error:

```
pip install -U apio
```

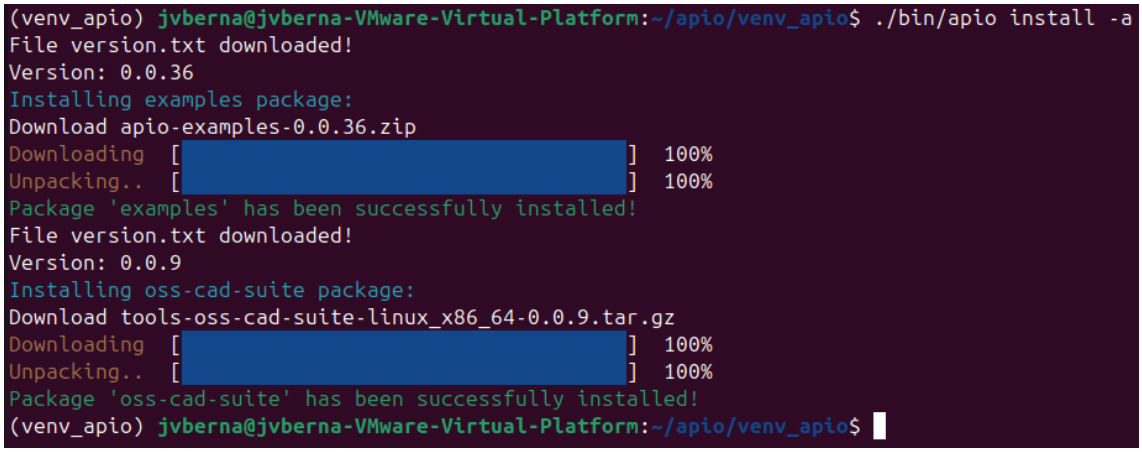
Aquí si ejecutas apio (simplemente lanza el comando APIO en la consola) verás que ya está instalado. Deberías ver algo parecido a esto:



4.- Instala todo el tool-chain de APIO

```
apio install -a
```

Con esto se instalarán odas las herramientas necesarias. El resultado debería ser similar a esto:



5.- Desactiva el Entorno: Una vez que hayas terminado de trabajar, puedes salir del entorno con:

```
deactivate
```

Recuerda que deberás reactivar el entorno (Paso 2) cada vez que reinicies tu terminal para usar apio.

Simular un ejemplo

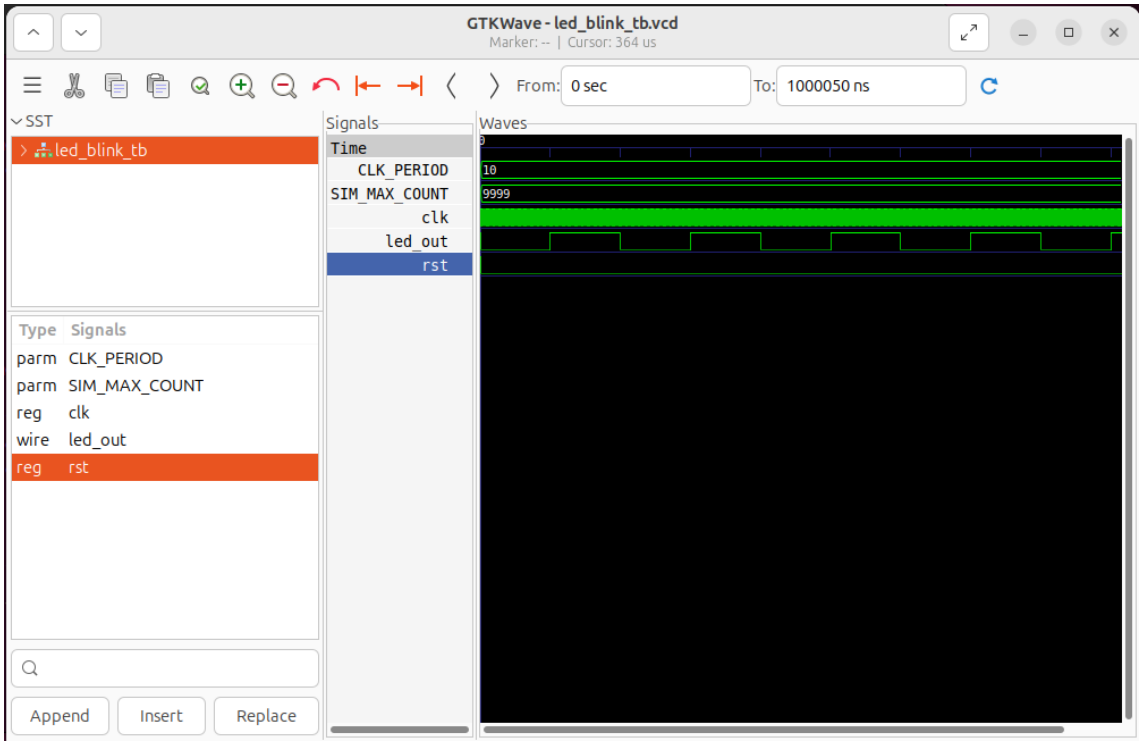
Descarga el código de ejemplo “Ejemplo LED verilog” que hay en el tema 8. Descomprímelo en una carpeta. Recuerda que para poder utilizar APIO debemos tener activo el entorno virtual si hemos instalado APIO a través del entorno virtual.

Una vez descargado, este ejemplo está listo tanto para ser simulado como para ser sintetizado sobre la tarjeta iCEZUM Alhambra.

Para simular el ejemplo ejecuta en la carpeta donde has descomprimido el código:

```
apio sim
```

El resultado debería ser similar a esto:



En la simulación se abr el programa GTKWave mostrando las señales que se han producido en el programa.

Error gtkwve: not found

Este error se produce porque el software gtkwave no está instalado en su sistema. Verás algo similar a esto:

```
(venv_apio) jvberna@jvberna-VMware-Virtual-Platform:~/apio/venv_apio/icezum/leds$ apio sim
gtkwave --rcvar "splash_disable on" --rcvar "do_initial_zoom_fit 1" leds_tb.vcd leds_tb.gtkw
sh: 1: gtkwave: not found
scons: *** [sim] Error 127
```

Para solucionarlo debes instalar el software, para ello ejecuta lo siguiente:

```
sudo apt update

sudo apt install gtkwave
```

Ahora ya no deberías tener problemas para hacer la simulación, prueba de nuevo “apio sim”.

Sintetizar y subir a la FPGA

Para que el hardware finalmente sea configurado en la FPGA se han de realizar dos pasos, construir y subir el ejecutable. Para ello ejecutamos:

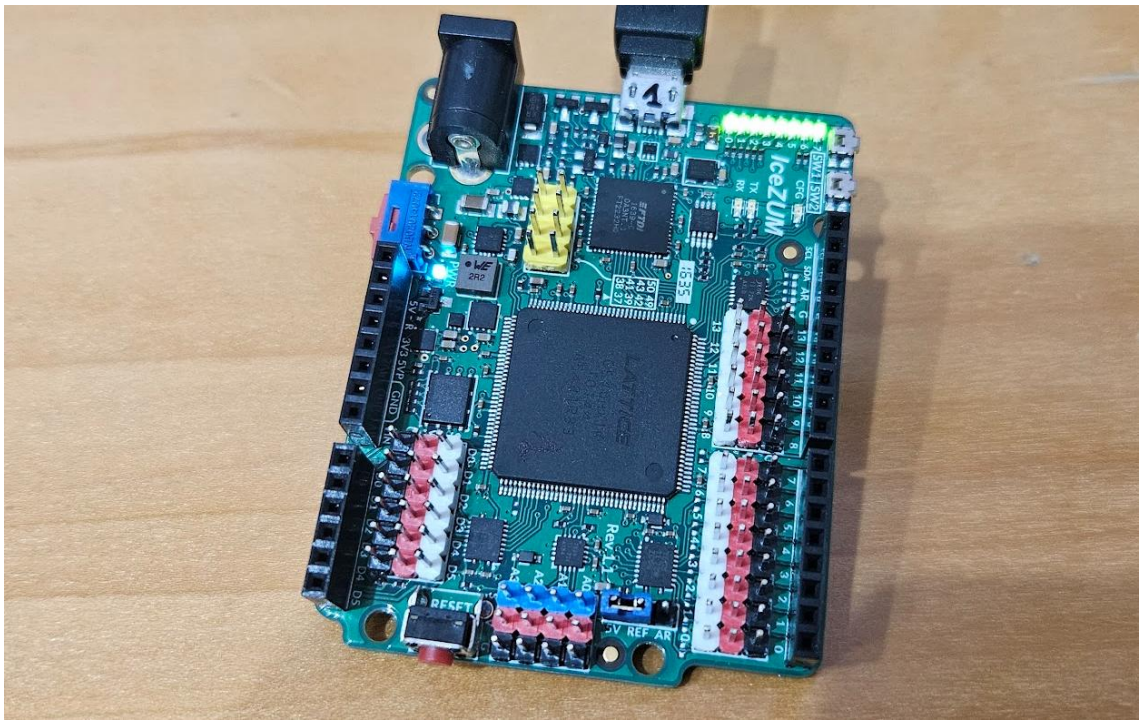
```
apio build
```

Deberíamos ver algo similar a esto:

```
(venv_apio) jvberna@jvberna-VMware-Virtual-Platform:~/apio/venv_apio/eje_clase/ejemplo$ apio build
[Thu Nov 13 11:11:55 2025] Processing icezum
--------------------------------------------------------------------------------------------------------
yosys -p "synth_ice40 -top led_blink -json hardware.json" -q led_blink.v
nextpnr-ice40 --hx1k --package tq144 --json hardware.json --asc hardware.asc --pcf led_blink.pcf -q
icepack hardware.asc hardware.bin
================================== [SUCCESS] Took 2.46 seconds ==================================
(venv_apio) jvberna@jvberna-VMware-Virtual-Platform:~/apio/venv_apio/eje_clase/ejemplo$ ls
abc.history  hardware.asc  hardware.json  led_blink_tb.out  led_blink_tb.vcd
apio.ini     hardware.bin  led_blink.pcf  led_blink_tb.v    led_blink.v
```

Esto generará varios archivos más, entre ellos un .bin que contiene el ejecutable de hardware.

Conectamos la plaza a nuestro puerto USB, y veremos como la placa iCEZUM Alhambra se enciende.



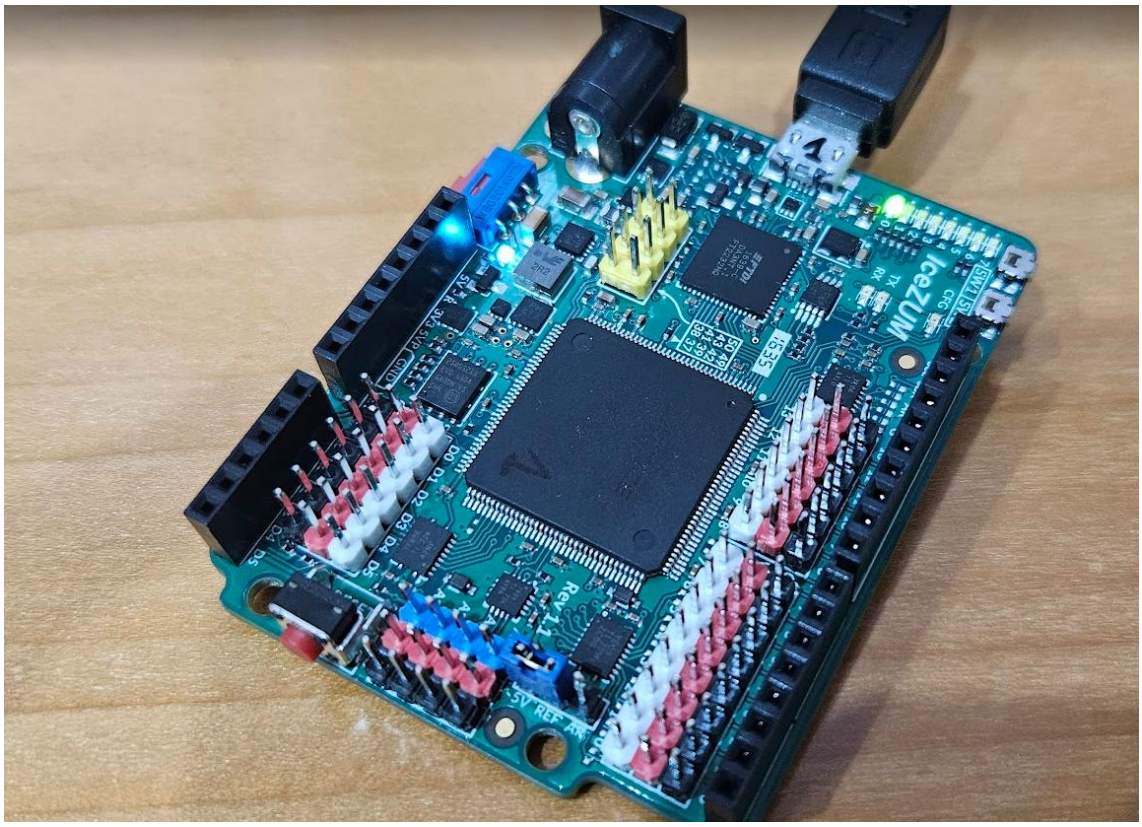
Ejecutamos el comando para cargar el programa en la FPGA.

```
apio upload
```

Tras esto deberíamos ver un proceso de carga en la plaza y finalmente un resultado como este:

```
(venv_apio) jvberna@jvberna-VMware-Virtual-Platform:~/apio/venv_apio/eje_clase/ejemplo$ apio upload
[Thu Nov 13 11:14:26 2025] Processing icezum
--------------------------------------------------------------------------------------------------------
iceprog -d i:0x0403:0x6010:0 hardware.bin
init..
cdone: high
reset..
cdone: low
flash ID: 0x20 0xBA 0x16 0x10 0x00 0x00 0x23 0x54 0x82 0x46 0x06 0x00 0x96 0x00 0x22 0x19 0x01 0x16
 0x3A 0x4B
file size: 32220
erase 64kB sector at 0x000000..
done.
VERIFY OK
cdone: high
Bye.
================================== [SUCCESS] Took 16.22 seconds ==================================
```

Tras la carga debería comenzar a parpadear el LED 1.



Error al hacer upload

Al conectar la FPA puede haber error debido al nivel de permisos para acceder al USB. Verás en pantalla un error como este: **libusb_open() failed**

El error **"ftdi_usb_get_strings failed: -4 (libusb_open() failed) / Error executing lsftdi"** indica un problema de comunicación entre la herramienta de carga (iceprog) y la placa iCEZUM a través del puerto USB. Específicamente, significa que la librería **libusb** (que maneja los dispositivos USB de bajo nivel) no pudo abrir el dispositivo FTDI.

Este error es casi siempre un **problema de permisos de usuario** en Linux (Ubuntu). Tu usuario no tiene permiso para acceder directamente al dispositivo USB de la FPGA.

Solución: Configurar Reglas UDEV (Recomendado)

La forma estándar y permanente de solucionar problemas de permisos de dispositivos USB es añadiendo reglas **udev**. Estas reglas le dicen a Linux que permita a ciertos grupos de usuarios (como el grupo plugdev) acceder a dispositivos específicos (como el adaptador FTDI que usa tu placa).

Sigue estos pasos:

1.- **Crea el archivo de reglas UDEV:** Abre tu terminal y crea o edita el archivo de reglas para 99-ftdi.rules:

```
sudo nano /etc/udev/rules.d/99-ftdi.rules
```

2.- **Pega las Reglas FTDI:** Copia y pega el siguiente contenido. Estas reglas permiten acceso a los dispositivos FTDI comúnmente usados en las placas FPGA.

Fragmento de código

```
# FTDI devices (needed for iceprog and open-ocd)

SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6010", MODE="0664", GROUP="plugdev"

SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6011", MODE="0664", GROUP="plugdev"

SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6014", MODE="0664", GROUP="plugdev"

# Añadido para otros dispositivos comunes

SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6015", MODE="0664", GROUP="plugdev"
```

Guarda el archivo (Ctrl+O, Enter) y ciérralo (Ctrl+X).

3.- **Añade tu Usuario al Grupo plugdev:** Asegúrate de que tu usuario pertenece al grupo que acabamos de autorizar (plugdev). Reemplaza $USER con tu nombre de usuario si no estás seguro.

```
sudo usermod -a -G plugdev $USER
```

4.- **Aplica los Cambios:** Recarga las reglas udev y desconecta/vuelve a conectar la placa:

```
sudo udevadm control --reload-rules
```

Desconecta y vuelve a conectar tu placa iCEZUM para que las nuevas reglas se apliquen.

5.- **Reinicia la Sesión:** Para que el cambio de grupo (usermod) tenga efecto, **debes cerrar la sesión de tu terminal o reiniciar tu computadora**.

Después de reiniciar la sesión, prueba “apio upload” de nuevo. El error de libusb_open() debería desaparecer.

Instalación de APIO en Windows

En el entorno Windows también es posible instalar APIO y para ello podemos acceder a la web del proyecto y revisar sus procesos:

https://github.com/FPGAwars/apio

En la parte inferior podremos encontrar acceso a la documentación:

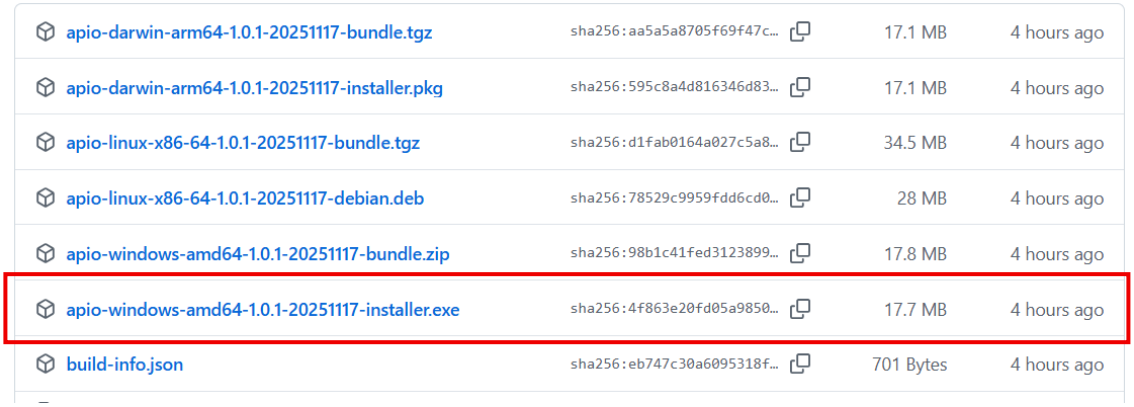
https://fpgawars.github.io/apio/docs/

Y dentro de la documentación, en la sección “Quick start” podremos encontrar las opciones de instalación.

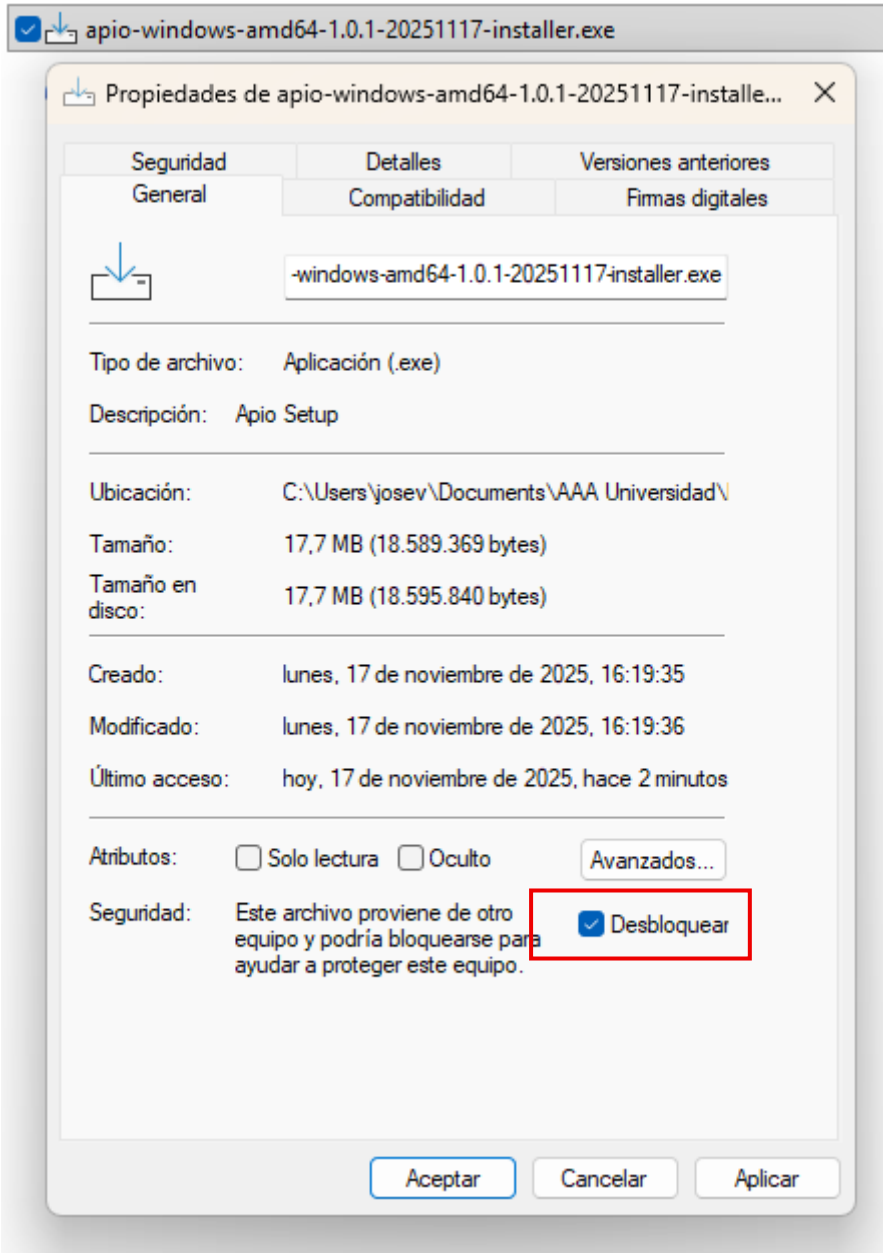
https://fpgawars.github.io/apio/docs/installing-apio/

Para el entorno Windows lo más sencillo es utilizar el clásico instalador, que podrás encontrar en la última reléase:

https://github.com/fpgawars/apio-dev-builds/releases

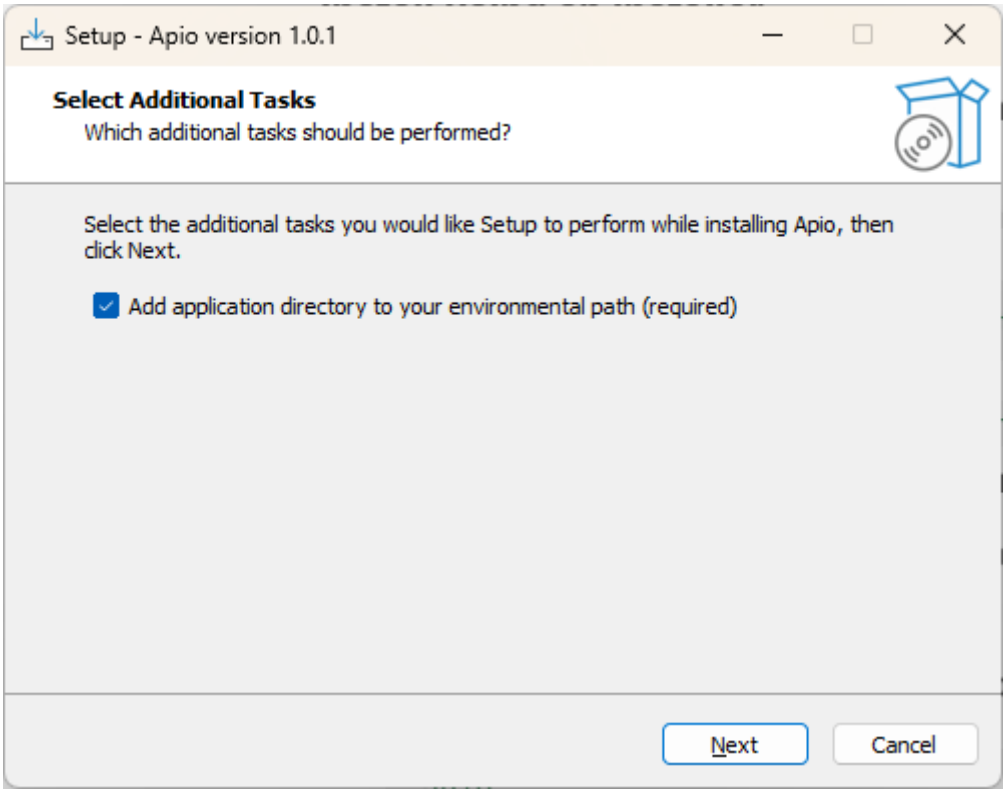


Una vez que descargas el instalador, debes activar una opción denominada “Desbloquear” que encontrarás dentro de la pestaña General:



Esto se debe a que el instalador no está firmado, y esta opción te permitirá ejecutar.

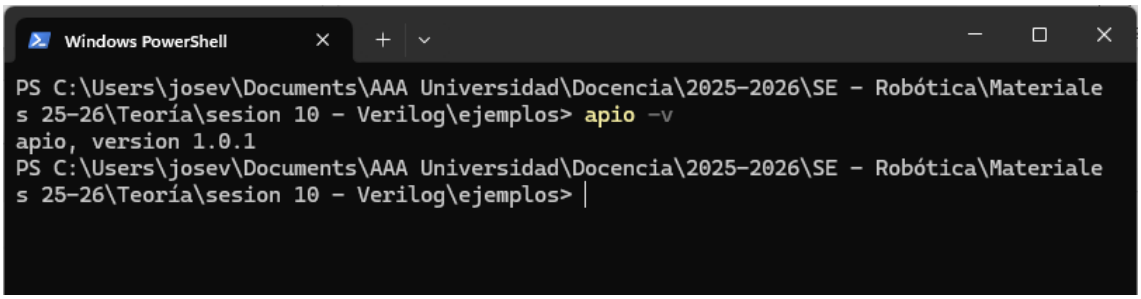
Al arrancar el instalador, debes seguir el proceso paso a paso: recuerda dejar marcada la opción “Add application directory to your environment path” para que una vez instalado, APIO funcione sin problemas.



Cuando haya finalizado la instalación, en la línea de comandos escribe el siguiente comando para comprobar que ha funcionado correctamente la instalación:

```
apio -v
```

Deberías ver algo similar a esto:

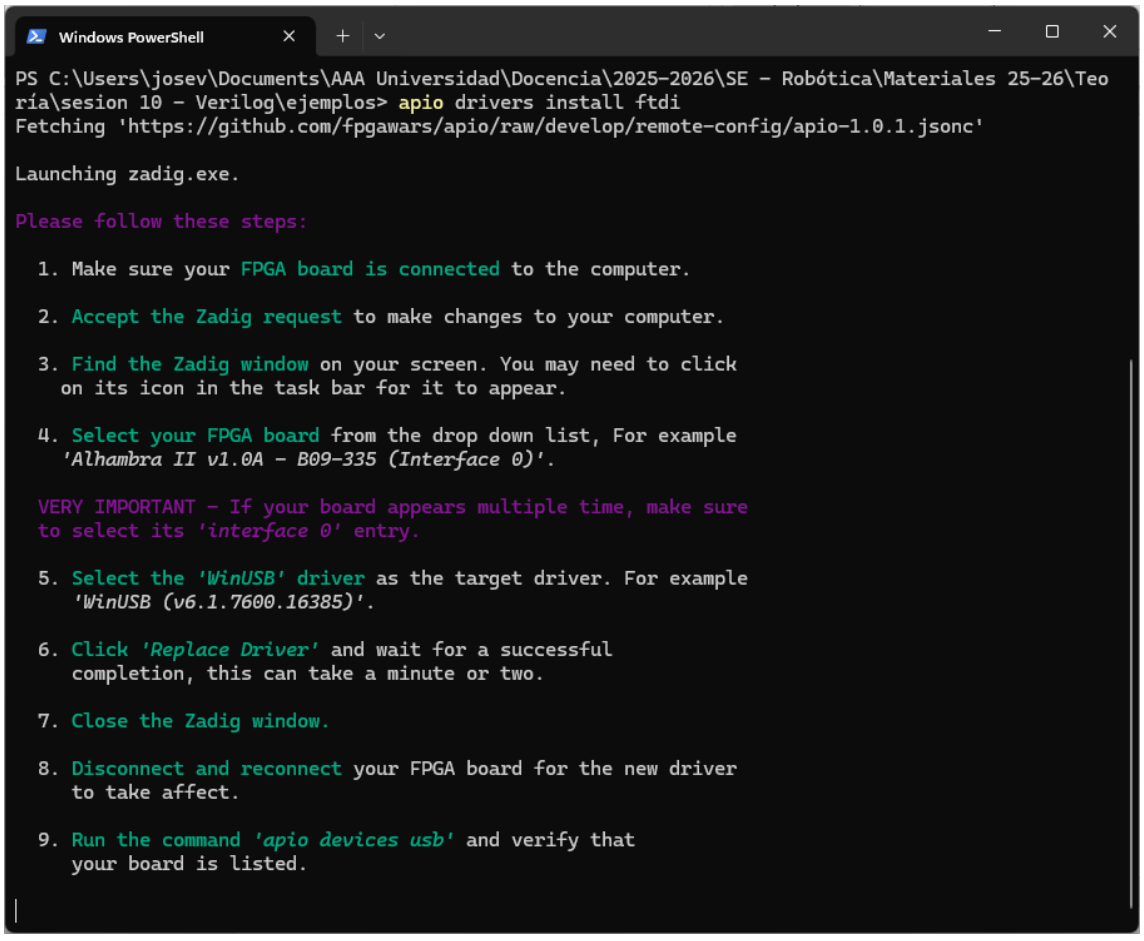


Instalación de drivers para conectar la plaza FPGA ICEZUM / ALHAMBRA II

Para poder cargar los programas que compilemos en nuestra placa FPGA es necesario instalar los drivers de la plaza en nuestro sistema operativo. Para ello, una vez instalado APIO debemos ejecutar lo siguiente:

```
apio drivers install ftdi
```

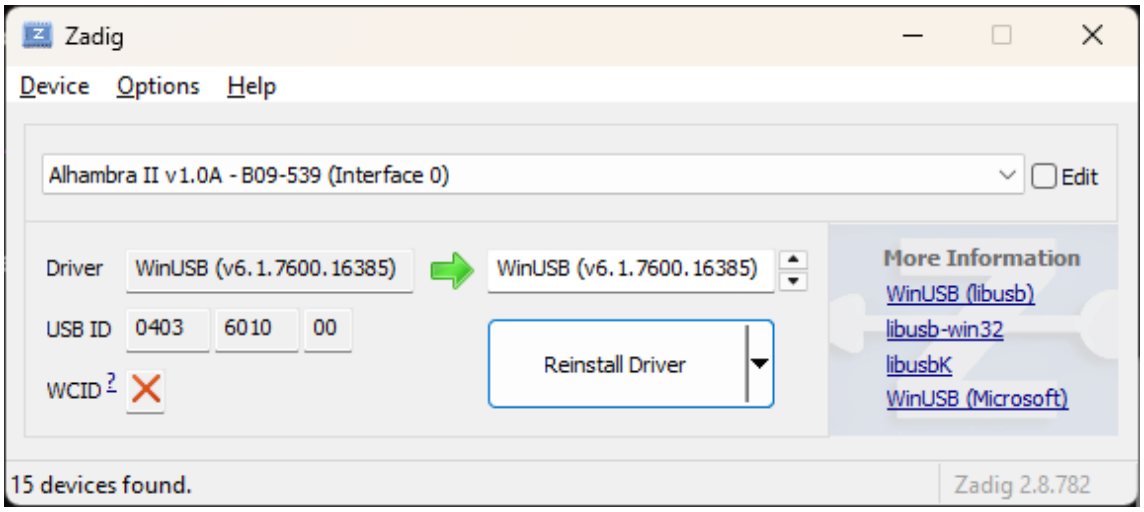
Esto mostrará una Ventana con las siguientes instrucciones (debes seguir las instrucciones que muestre tu ejecución)



Como ves, indica una secuencia de pasos donde debes:

- Conectar tu placa FPGA (esto ayudará al software a detectar el driver a utilizar), arrancará el controlador de drivers, instala el driver en el puerto donde aparece la ICEZUM.

Arrancará la aplicación Zadig, de instalación de drivers. Selecciona tu tarjeta y pulsa el botón “Reinstal Driver”.



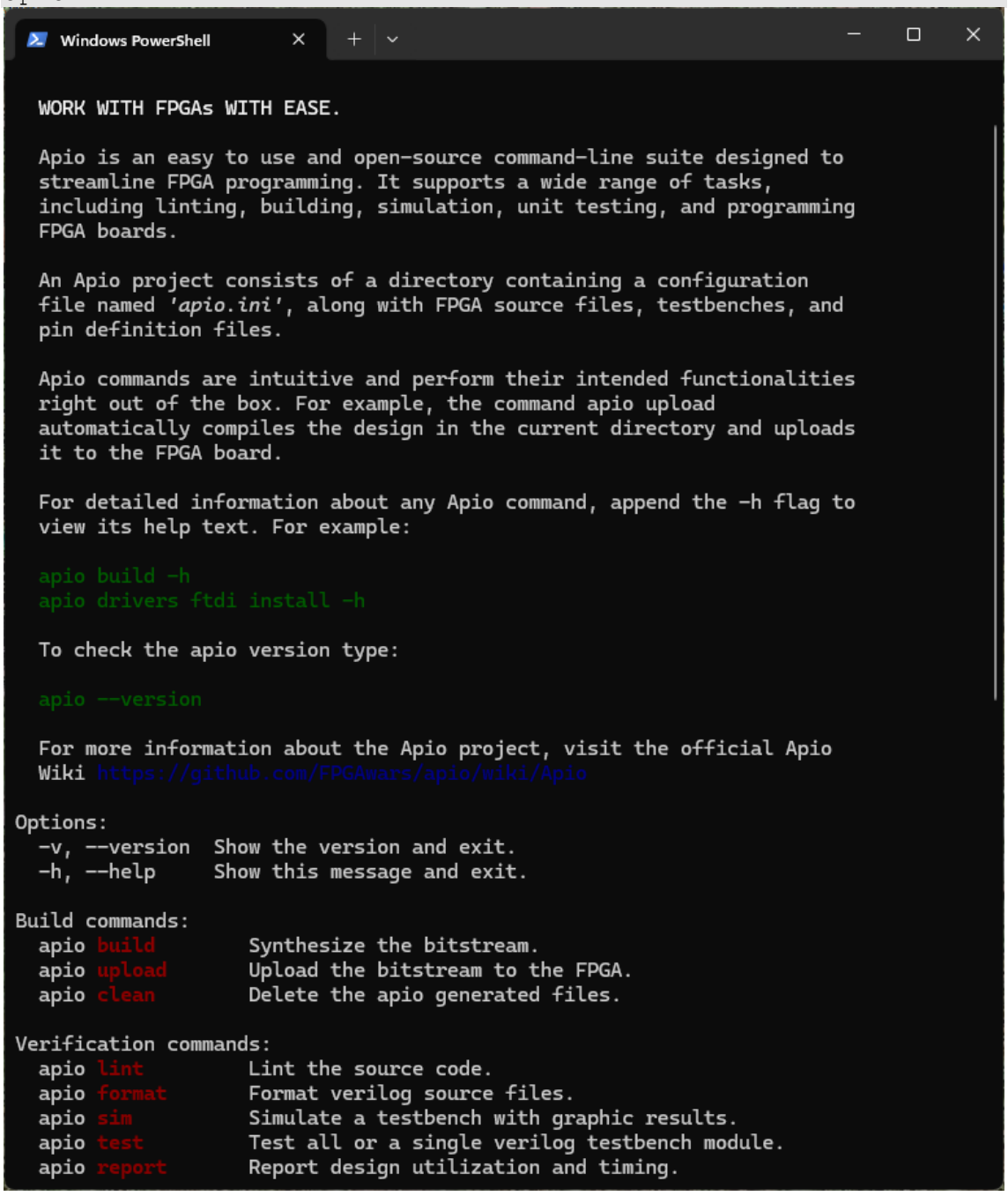
Nota: aquí se muestra el driver para Alhambra II, para la versión 1.1 del laboratorio aparecerá ICEZUM 1.1

Comandos de APIO

Veamos algunos comandos importanes de APIO.

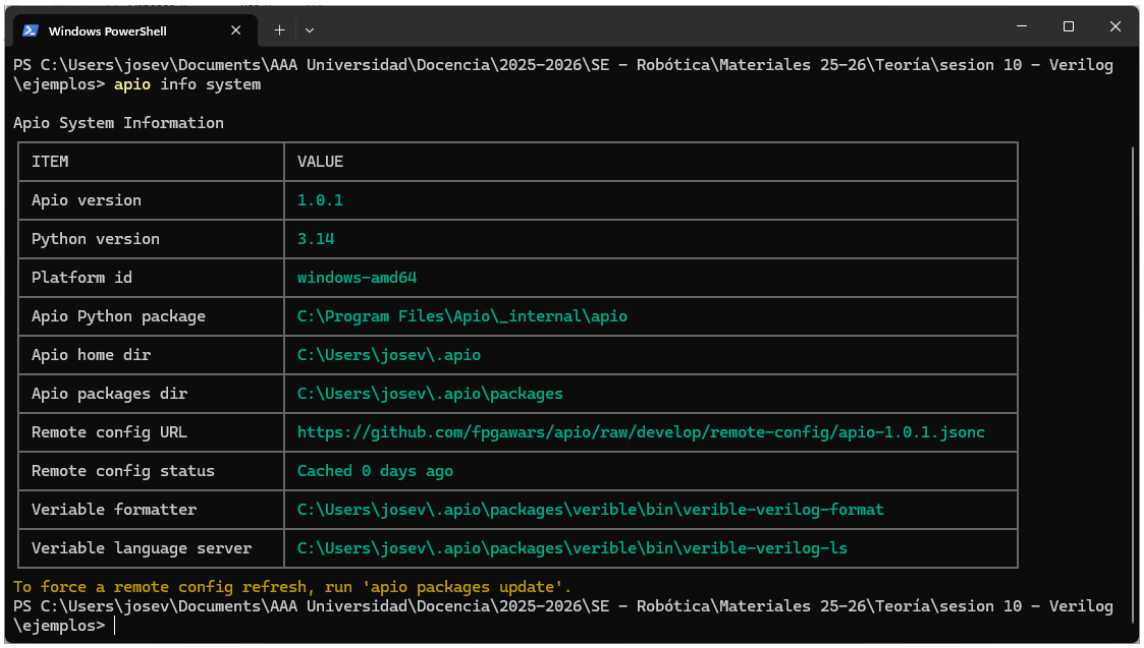
Ayuda de APIO, para listar todos los comandos y opciones:

```
apio -h
```



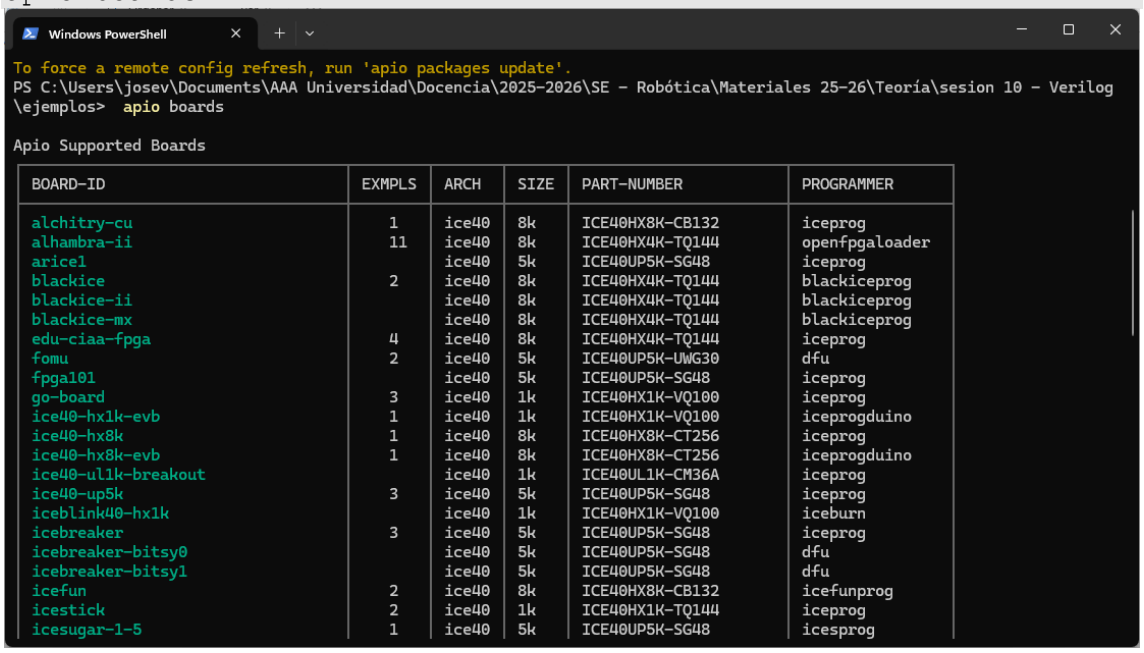
Para obtener información del sistema:

```
apio info system
```



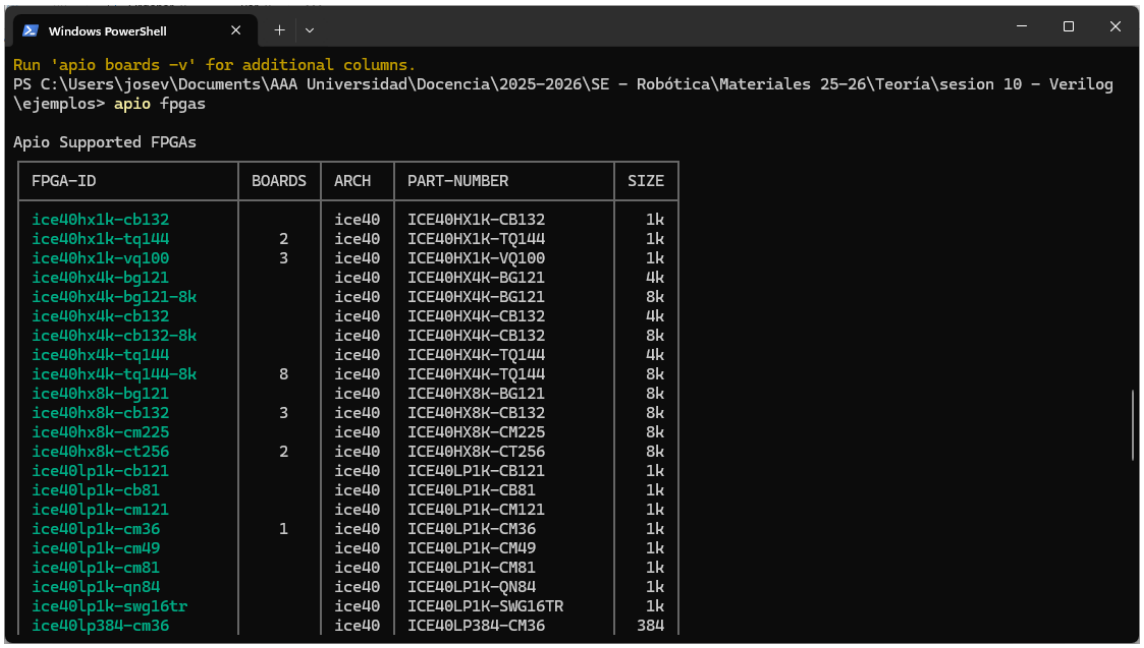
Si queremos conocer que placas son soportadas por APIO:

```
apio boards
```



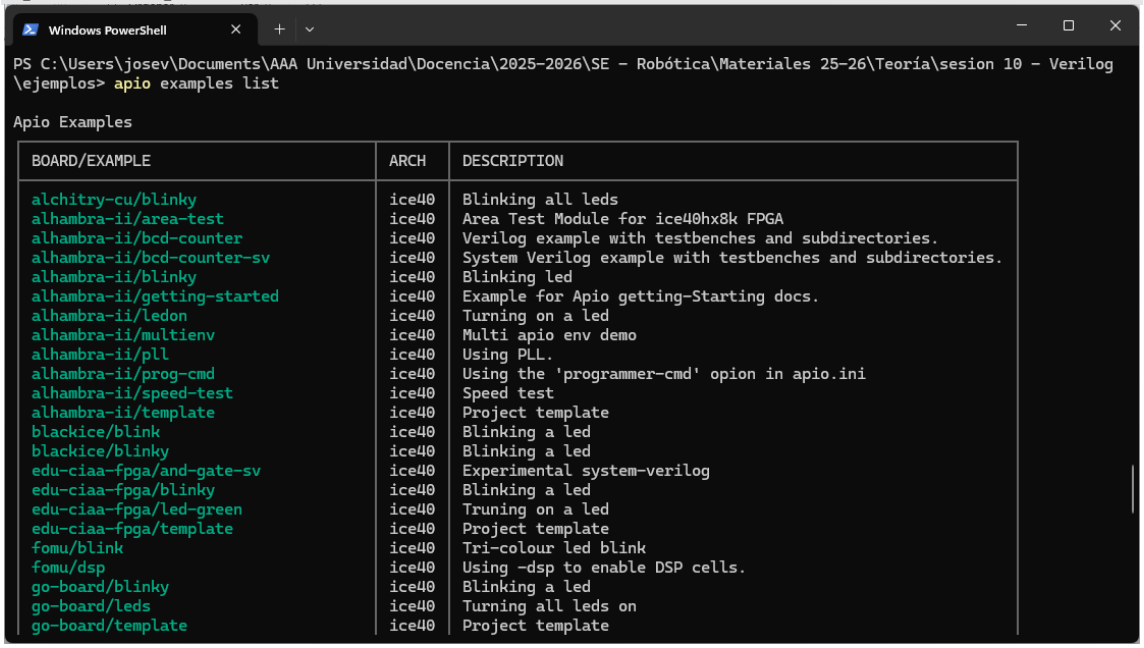
Y si queremos conocer las FPGA que soporta APIO:

```
apio fpgas
```



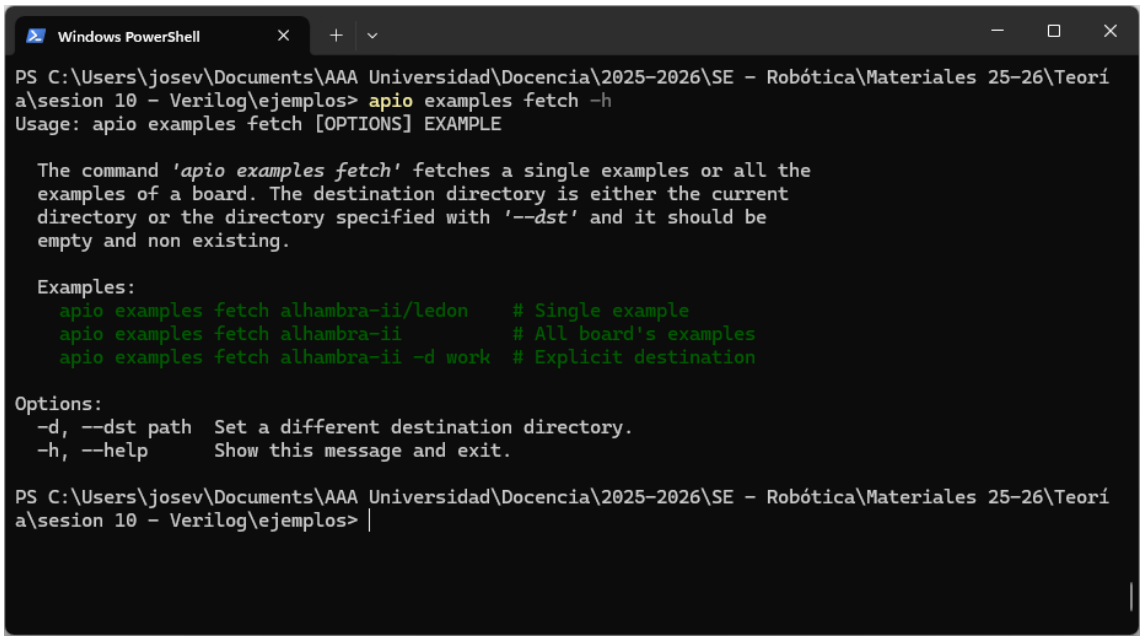
Una buena forma de comenzar a trabajar en Verilog, es precisamente utilizando ejemplos. APIO trae por defecto muchos ejemplos de distintas placas, podemos ver la lista de ejemplos con este comando:

```
apio examples list
```



Y para instalar ejemplos utilizamos el comando fetch. Para saber más sobre su uso podemos introducir:

```
apio examples fetch -h
```



Tal como se muestra en el resultado de la ayuda, podemos instalar todos los ejemplos de un tipo de placa, o un ejemplo en concreto. Se instalan por defecto en una carpeta con el mismo nombre que la placa. Por ejemplo, con el siguiente comando se instalan todos los ejemplos de la placa ICEZUM, en un directorio denominado “icezum”:

```
pio examples fetch icezum -d icezum
```

Verilog

Verilog es un lenguaje de descripción de hardware (HDL) usado para modelar circuitos digitales. A diferencia de un lenguaje de programación tradicional, con Verilog describes hardware real: compuertas, registros, memorias, módulos, etc.

El bloque fundamental de Verilog es el módulo.

```
module nombre_modulo (entradas, salidas);
// Declaración de puertos
// Lógica interna
endmodule
```

Ejemplo simple:

```
module and_simple(
    input wire a,
    input wire b,
    output wire y
  );
  assign y = a & b;
endmodule
```

Tipos de señales

Los dos tipos principales:

wire

- Representa una conexión física.
- Se usa para conexiones combinacionales.
- No puede almacenar estado.

reg

- Puede guardar un valor hasta que cambie.
- Se usa dentro de bloques always.

```
reg a;
wire b;
```

Operadores comunes

- & AND
- | OR
- ^ XOR
- ~ NOT
- +, -, * aritmética

- ==, != comparaciones

Bloques always

Se usan para lógica secuencial o combinacional.

Lógica combinacional:

```
always @(*) begin
    y = a & b;
end
```

Lógica secuencial (con reloj):

```
always @(posedge clk) begin
    q <= d;
end
```

Asignaciones

Asignación continua (combinacional):

```
assign y = a ^ b;
```

Asignación en bloque always:

= asignación bloqueante (combinacional)

<= asignación no bloqueante (secuencial)

Flancos de reloj

`posedge clk` significa flanco positivo del reloj ("positive edge of clock"). En Verilog, se usa dentro de un bloque always para indicar que la lógica debe ejecutarse solo cuando la señal `clk` pasa de 0 a 1.

Una señal de reloj (clk) es una onda cuadrada que alterna entre 0 y 1, cada vez que la señal sube de 0 → 1 ocurre un **flanco positivo**.

```
__|‾‾|__|‾‾|__|‾‾|__
  ↑    ↑    ↑
  flanco positivo
```

Es importante porque en hardware digital, la mayoría de los elementos secuenciales (como registros y flip-flops) actualizan su estado solo en ese instante. Ejemplo típico:

```
always @(posedge clk) begin
    q <= d;
end
```

Significa:

- Cada vez que llega un flanco positivo del reloj
- El valor de d se guarda en q

Esto es exactamente cómo funciona un flip-flop D. Se pueden utilizar el flanco positivo o el negativo (es muy infrecuente el negativo), necesarios para describir lógica secuencial en Verilog:

- posedge = flanco ascendente
- negedge = flanco descendente

Assign

`assign` se usa para hacer asignaciones continuas, es decir, para describir lógica combinacional que está siempre activa, como si fueran cables y compuertas. Es de las instrucciones más importantes del lenguaje.

```
assign y = a & b;
```

Esto describe una compuerta AND:

```
a ----\
       )--- y
b ----/
```

No espera un reloj, ni un flanco, ni un bloque always.

Assing debe usarse cuando:

- Para lógica combinacional simple
- Para conexiones directas entre señales
- Para operaciones aritméticas o lógicas que no necesitan memoria
- Para buses y señales que deben actualizarse de forma continua

Por tanto, solo debe usarse con `wire` o tipos derivados (`tri`, `wand`, etc). **Nunca** se usa con `reg` y **siempre** está activa, no "se ejecuta" como en un programa.

```
assign s = a ^ b;
assign sum = x + y;
assign out = in;
assign y = sel ? b : a;  // multiplexor simple
```

Nunca se debe usar con lógica secuencial, es decir, que dependa de un ciclo de reloj utilizando bloques `always @(posedge clk)`.

Always

`always` se usa para describir comportamiento del hardware, es decir, cómo cambia un circuito cuando cambian sus señales, permite escribir:

- **Lógica combinacional más compleja**
- **Lógica secuencial (con reloj)**
- **Estados, contadores, máquinas de estados, etc.**

Lógica combinacional equivale a describir cómo se comportan salidas en función de señales de entrada, pero no guarda datos.

```
always @(*) begin
    y = a & b;   // lógica combinacional
end
```

Lógica secuencial implica el uso de ciclo de reloj, y se usa para el almacenamiento de datos.

```
always @(posedge clk) begin
    q <= d;   // lógica secuencial (flip-flop)
end
```

Verilog describe **hardware real**, no software.

- Un always @(*) equivale a compuertas lógicas.
- Un always @(posedge clk) equivale a flip-flops en un circuito.

Bloqueante y no bloqueante

En Verilog, dentro de un bloque always, las asignaciones pueden ser **bloqueantes (=)** o **no bloqueantes (<=)**. Esto es muy importante porque afecta cómo se *simula* y cómo se *sintetiza* el hardware.

Asignación bloqueante =

Significa que la instrucción se ejecuta inmediatamente y bloquea las siguientes líneas hasta terminar. Se usa normalmente para lógica combinacional. Ejemplo:

```
always @(*) begin
    a = b;
    c = a;   // aquí 'a' ya tiene el nuevo valor de 'b'
end
```

Flujo de ejecución:

1. a = b se ejecuta **ya**

2. c = a usa el valor recién asignado

Esto funciona como instrucciones en un lenguaje de programación tradicional.

Asignación no bloqueante <=

Significa que no se actualiza inmediatamente. Se programa el cambio para el final del ciclo de simulación. Se usa para lógica secuencial (en posedge clk). Ejemplo:

```
always @(posedge clk) begin
    a <= b;
    c <= a;   // aquí 'a' todavía NO ha tomado su nuevo valor
end
```

Flujo:

1. a <= b **se programa para actualizarse después**
2. c <= a usa el valor viejo de a
3. Al final del posedge, ambos valores cambian a la vez

Esto imita cómo funcionan los **flip-flops** en hardware real: *todos cambian simultáneamente al flanco del reloj*.

Un ejemplo que estaría mal es este: **Usando = (incorrecto para secuencial)**

```
always @(posedge clk) begin
    a = b;
    c = a;
end
```

Esto produce un comportamiento que **no existe en hardware real**, porque en simulación:

- a cambia inmediatamente
- c recibe este nuevo valor

El sintetizador podría darte un diseño incorrecto ya que se está haciendo la acción en un flanco de reloj.

Se debería usar: **<= (correcto)**

```
always @(posedge clk) begin
    a <= b;
    c <= a;
end
```

Esto produce **dos flip-flops en serie**, que es el comportamiento deseado.

```
b → [FF1:a] → [FF2:c]
```

Regla de oro (muy importante)

Tipo de lógica	Tipo de asignación recomendada
Secuencial (always @(posedge clk))	<= no bloqueante
Combinacional (always @(*))	= bloqueante

`=` → actualiza ahora, línea por línea

`<=` → actualiza todos juntos, al final del ciclo

Los flip-flops funcionan como `<=`

Las compuertas funcionan como `=`

Notas sobre la última actualización en APIO. Algunos comandos han cambiado de la versión 0.9 a la versión 1, igual que dentro de apio.ini pueden haber cambiado algunos elementos.

Código de los ejemplos

Todo el código que será utilizado en los ejemplos a continuación se encuentra publicado en el repositorio:

https://github.com/jvberna/se_verilog_apio_code_examples

Ejemplo 1 – Puerta AND

El siguiente ejemplo muestra en Verilog como implementar una puerta AND. Aprovecharemos para comentar todos los archivos necesarios para realizar una simulación.

El archivo principal tiene se denomina `and_assign.v` y tiene el siguiente contenido:

```
1 module and_assign(
2     input wire a,
3     input wire b,
4     output wire y
5 );
6 assign y = a & b;
7 endmodule
```

Analizando cada línea del código encontramos lo siguiente:

Línea 1: `module and_assign(` Comienza la definición de un módulo (la unidad fundamental en Verilog). El nombre de este módulo es **and_assign**.

Línea 2: `input wire a,` Declara la señal **a** como un puerto de entrada (input). La palabra clave **wire** indica que es un tipo de dato para conectar elementos (como un cable), el valor por defecto para entradas/salidas.

Línea 3: `input wire b,` Declara la señal b como un segundo puerto de entrada.

Línea 4: `output wire y` Declara la señal y como un puerto de salida (output). Esta será la salida del circuito.

Línea 5: `);` Cierra la lista de puertos del módulo.

Línea 6: `assign y = a & b;` Esta es la parte central, una asignación continua. Utiliza la palabra clave assign para conectar permanentemente la salida **y** al resultado de la expresión lógica AND (&) de las entradas a y b. Esto modela directamente el comportamiento de una compuerta AND.

Línea 7: `endmodule` Marca el final de la definición del módulo and_assign.

Además de este archivo es necesario crear un archivo .ini que indicará ciertos aspectos de la simulación o la síntesis. Para ello utilizamos:

```
apio create -b icezum
```

Esto crea un archivo apio.ini con la siguiente estructura.

```
# APIO project configuration file.
```

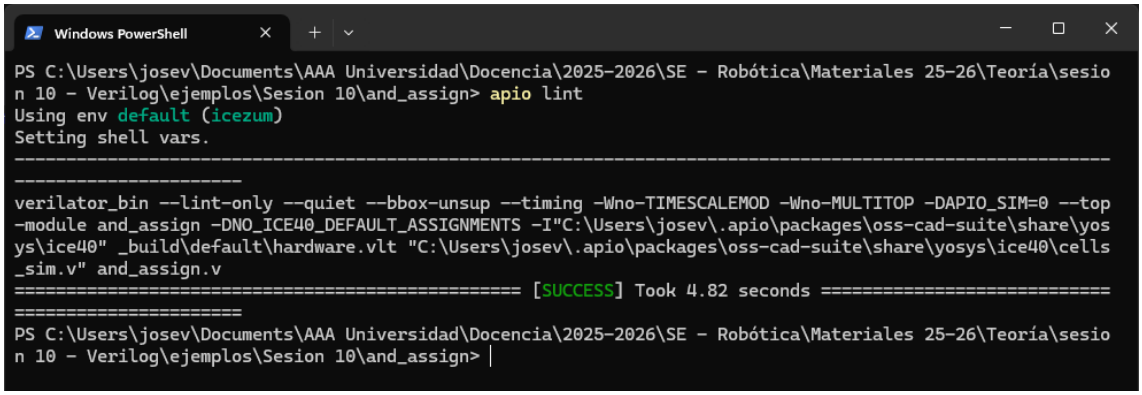
```
# For details see https://fpgawars.github.io/apio/docs/project-file

[env:default]
board = icezum
top-module = main
```

Se ha de indicar la placa a la que está destinada la simulación o síntesis porque de ella dependerá cómo se cree el circuito interno. Además de la placa (board) vemos que hay una línea “top-model” que indica cual es el módulo de nivel superior, y le dice a la herramienta de síntesis (en nuestro caso Yosys) el punto de entrada del diseño.

Aquí vamos a introducir el primer cambio, ya que nuestro módulo no se llama “main” sino “and_assign”, por lo que apio.ini quedará de esta forma:

```
[env:default]
board = icezum
top-module = and_assign
```

Ahora Podemos usar la comprobación de código “apio lint” para comprobar que el código es correcto. El resultado de ejecutar “lint” será algo similar a esto:



El siguiente paso, es generar la simulación del circuito. Para ello es necesario crear un archivo más, a través del cual se generen señales que ayuden a simular el funcionamiento del circuito. En nuestro caso el circuito se trata de dos entradas “a” y “b” que generan una puerta AND sobre la salid “y”.

El archivo a través del cual se simulan señales de entrada se llama “testbench”. Habitualmente se crea con el mismo nombre que le archivo verilog, pero acabado en “_tb.v”. En nuestro caso vamos a crear `and_assign_tb.v` con el siguiente contenido:

```
`timescale 1ns/1ps

module and_assign_tb;

reg a;
reg b;
```

```
wire y;

and_assign uut (
    .a(a),
    .b(b),
    .y(y)
);

initial begin
    $dumpfile("_build/default/and_assign_tb.vcd
    $dumpvars(0, and_assign_tb

    $display("Tiempo | a b | y");
    $monitor("%4t   | %b %b | %b", $time, a, b, y);

    // Estímulos
    a = 0; b = 0; #100;
    a = 0; b = 1; #100;
    a = 1; b = 0; #100;
    a = 1; b = 1; #100;

    $finish;
end

endmodule
```

Un Testbench tiene como propósito simular el funcionamiento de un módulo (Device Under Test o DUT) aplicando entradas y verificando salidas. Veamos línea a línea que es cada cosa.

`` `timescale 1ns/1ps `` → directiva de escala de tiempo, indica la unidad de tiempo para la simulación (1 nanosegundo) y la precisión del tiempo (1 picosegundo). La precisión siempre debe ser igual o menor que la unida de tiempo.

Hay que tener en cuenta que luego en la simulación se introducen retardos (representados por #100 por ejemplo), estos retardos se medirán en la unidad que se haya indicado en la escala del tiempo para simulación, por lo que #100 indica un retardo de 100 milisegundas.

Disminuir la precisión aumenta la exactitud, pero se empleará más memoria y CPU en la simulación.

Las opciones que puedes usar para la **unidad** y la **precisión** son potencias de 10:

- **Segundos:** s
- **Milisegundos:** ms, 10^{-3}s
- **Microsegundos:** us, 10^{-6}s
- **Nanosegundos:** ns, 10^{-9}s

- **Picosegundos:** ps, 10^{-12}s
- **Femtosegundos:** fs, 10^{-15}s

Habitualmente se emplea 1ns/1ps o 1ns/100ps, que ofrecen un buen equilibrio entre velocidad y fidelidad temporal.

`module and_assign_tb;` → en Verlog, todo se declara dentro de un module

`reg a; reg b;` →declaración de variables de tipo reg. Las señales que impulsan las entradas del DUT siempre deben ser `reg` en el testbench, porque sus valores deben ser almacenados y cambiados dentro del bloque `initial`.

`wire y;` → Señal de tipo wire, las señales que reciben la salida del DTU siempre deben ser `wire` en el testbench, ya que su valor es determinado continuamente por el DUT.

`and_assign uut (` → **Instancia el Módulo a Probar (DUT).** and_assign es el nombre de tu módulo. uut (Unit Under Test) es el nombre de esta instancia.

`.a(a),` → **Conexión de Puerto.** Conecta el puerto interno *.a* del DUT a la señal *a* declarada en el testbench. Igual para .b(b), y .y(y).

`initial begin` → **Bloque de Procedimiento Inicial.** Todo el código dentro de este bloque se ejecuta **una sola vez** al inicio de la simulación (t=0).

`$dumpfile("_build/default/and_assign_tb.vcd");` → **Tarea del Sistema.** Indica al simulador que cree un archivo con el nombre especificado (.vcd - Value Change Dump) para guardar los cambios de señal a lo largo del tiempo. Esto se usa para visualizadores de formas de onda (como GTKWave). Si no se indica un dumpfile se generaria por defecto con el nombre del testbench en la carpeta _build/default.

Todas las líneas que comienzan por **$** indican que es una tarea del sistema.

Nota, el archive vcd con las formas de ondas deberá generarse o bien en _build/default o en la raíz del proyecto, depende de la versión de APIO que estemos utilizando. Fíjate cuando hagas la simulación si GTK lanza algún error porque no encuentra el archivo.

`$dumpvars(0, and_assign_tb);` → **Tarea del Sistema.** Registra todas las variables y señales dentro del módulo and_assign_tb y sus submódulos (el DUT) para ser incluidas en el archivo .vcd. El 0 indica la profundidad de la jerarquía, en este caso todas las jerarquías. Es decir, se almacenarán todas las variables de todos los módulos.

`$display("Tiempo | a b | y");` → Tarea del sistema. Imprime texto por la consola de simulación.

`$monitor("%4t | %b %b | %b", $time, a, b, y);` → Otra tarea del sistema, funciona como un rastreador de cambios, se activa automáticamente y realiza la impresión de texto cada vez que una de las variables en su lista de argumentos cambia de valor.

`a = 0; b = 0; #100;` → Asigna los valores iniciales a las entradas (a=0, b=0). Luego, espera 100 ns, permitiendo que la salida y se estabilice y que el $monitor registre el resultado. Similar para el resto de estímulos.

`$finish;` → **Tarea del Sistema.** Detiene la simulación y sale del simulador.

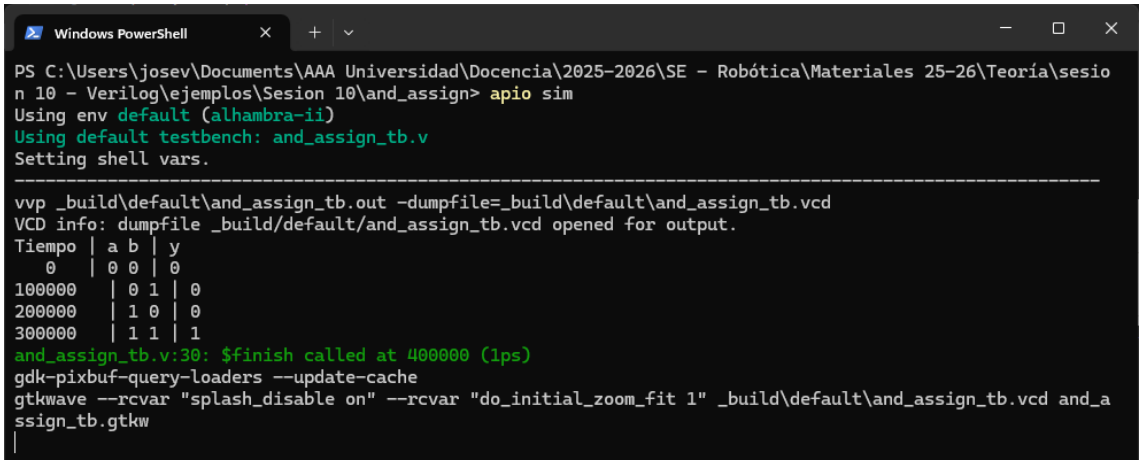
`end` → Cierra el bloque initial.

`endmodule` → Cierra el módulo and_assign_tb

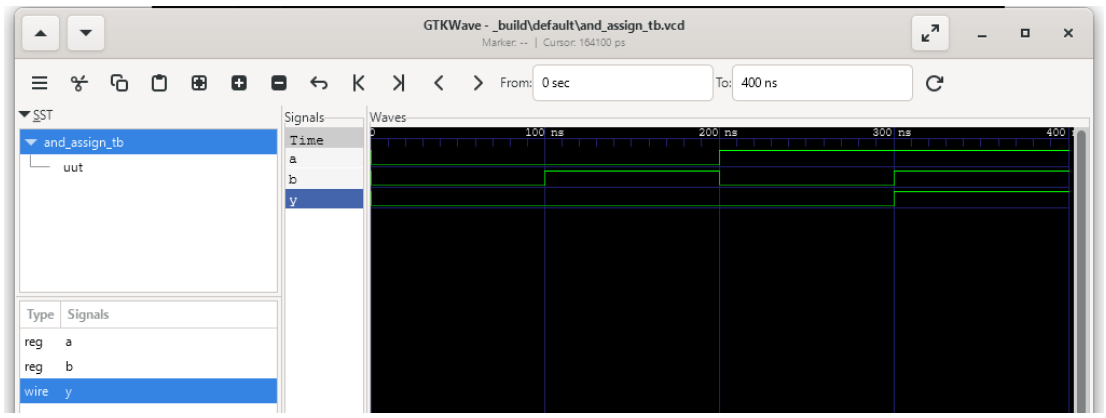
Ahora que ya tenemos claro que hace el testbench, podemos simular el circuito, para eso ejecutamos:

```
apio sim
```

Deberías ver algo como esto:



Y se abrirá el analizador de ondas GTWave. Para ver las señales simuladas, pulsamos sobre “and_assign_tb”, y si hacemos doble clock sobre la lista de señales “a, b, y” para que se añadan a al analizador. Veremos algo como esto:



Aunque esto es una puerta lógica, y en sí mismo no tiene sentido sintetizarla, podríamos probarla sobre nuestra placa de una forma sencilla, por ejemplo utilizando como entradas los botones que tiene nuestra Alhambra II, y como salida uno de los leds. Para ello es necesario crear el archivo de configura de pines, and_assign.pcf con las siguientes líneas:

```
set_io a 34
set_io b 33
set_io y 45
```

Ahora si haces apio build, y posteriormente apio upload, podrás ver el funcionamiento de la puerta AND utilizando los botones.

Ejemplo 2 - Leds

En este ejemplo vamos a encender los varios leds y a apagar otros. Para ello utilizaremos el siguiente código:

```
module leds (
    output wire LED0,
    output wire LED1,
    output wire LED2,
    output wire LED3,
    output wire LED4,
    output wire LED5,
    output wire LED6,
    output wire LED7
);

  assign LED0 = 1'b0;
  assign LED1 = 1'b1;
  assign LED2 = 1'b0;
  assign LED3 = 1'b1;
  assign LED4 = 1'b0;
  assign LED5 = 1'b1;
  assign LED6 = 1'b0;
  assign LED7 = 1'b1;

endmodule
```

El testbench será el siguiente:

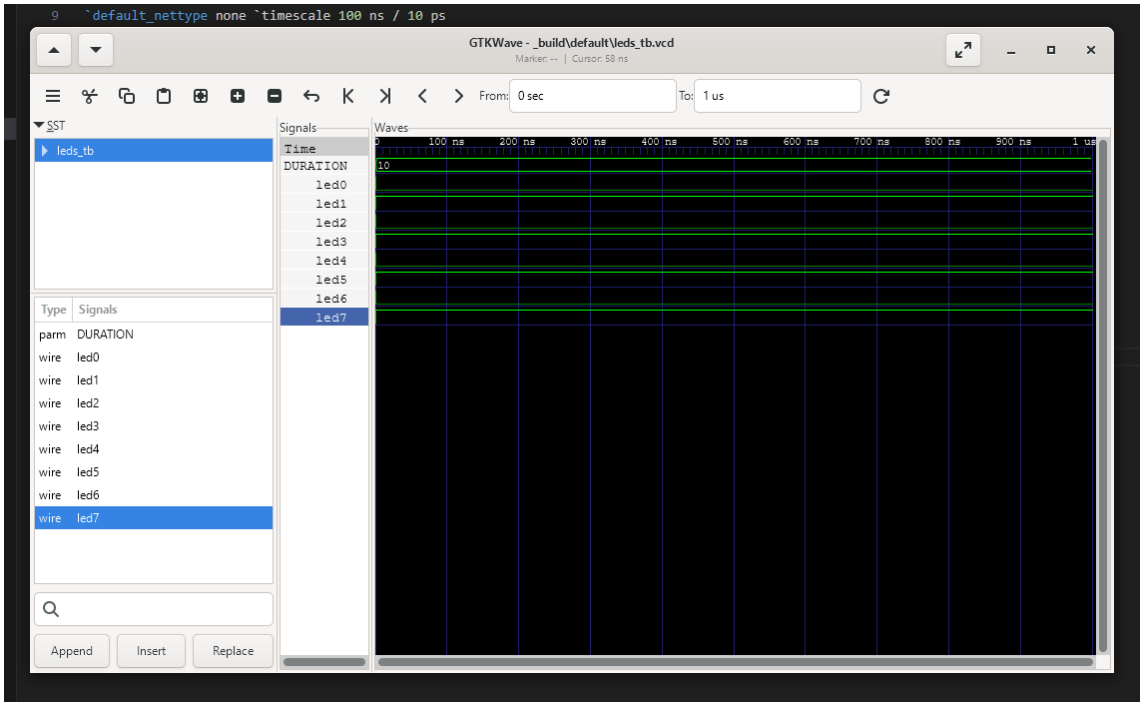
```
`default_nettype none `timescale 100 ns / 10 ps

module leds_tb ();

  parameter DURATION = 10; //-- Simulation time: 1us (10 * 100ns)

  //-- Leds port
  wire led0, led1, led2, led3, led4, led5, led6, led7;

  //-- Instantiate the unit to test
  leds UUT (
      .LED0(led0),
      .LED1(led1),
      .LED2(led2),
      .LED3(led3),
      .LED4(led4),
      .LED5(led5),
      .LED6(led6),
      .LED7(led7)
  );


  initial begin

    //-- Dump vars to the .vcd output file
    $dumpvars(0, leds_tb);
```

```
    #(DURATION) $display("End of simulation");
    $finish;
  end

endmodule
```

Y por último apio.ini

```
[env:default]
board = alhambra-ii
top-module = leds
default-testbench = leds_tb.v
```

Al realizar la simulación veremos algo como esto: `apio sim`



Si te das cuenta, en la línea de tiempo de la simulación se ha llegado hasta 1us (1 microsegundo). Esto se debe a que la parameter `DURATION = 10;` y este parámetro se utiliza como retardo en `#(DURATION) $display("End of simulation");`. Como la escala de tiempo está establecida a 100ns, eso significa que una duración de 10 * 100ns es igual a 1 microsegundo.

```
default_nettype none
```

Esta es una directiva de compilador que controla cómo el simulador o sintetizador maneja las señales (nets) que no han sido declaradas explícitamente (wire, reg, etc.).

- **Comportamiento Por Defecto:** Sin esta directiva, si usas una señal en tu código sin declararla (por ejemplo, assign c = a & b; y la señal c no está

declarada), Verilog asume por defecto que es una **wire**. Esto se conoce como **cable implícito** (implicit net declaration).

- **Efecto de none:** Al establecer default_nettype none, **desactivas** este comportamiento por defecto. Si intentas usar una señal sin declararla, el compilador generará un **error**.

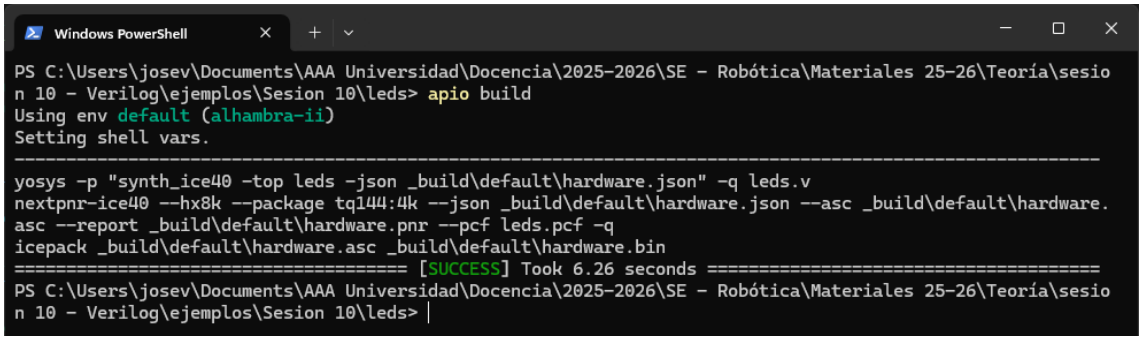
Se considera una **buena práctica de codificación** en HDL. Ayuda a:

- **Detectar Errores Tipográficos:** Si escribes mal un nombre de señal (p. ej., usas couner en lugar de counter), en lugar de crear un cable implícito (wire couner), el compilador te obligará a corregir el error tipográfico.
- **Mejorar la Claridad:** Obliga al desarrollador a declarar explícitamente el tipo de todas las señales, haciendo el código más claro y fácil de mantener.

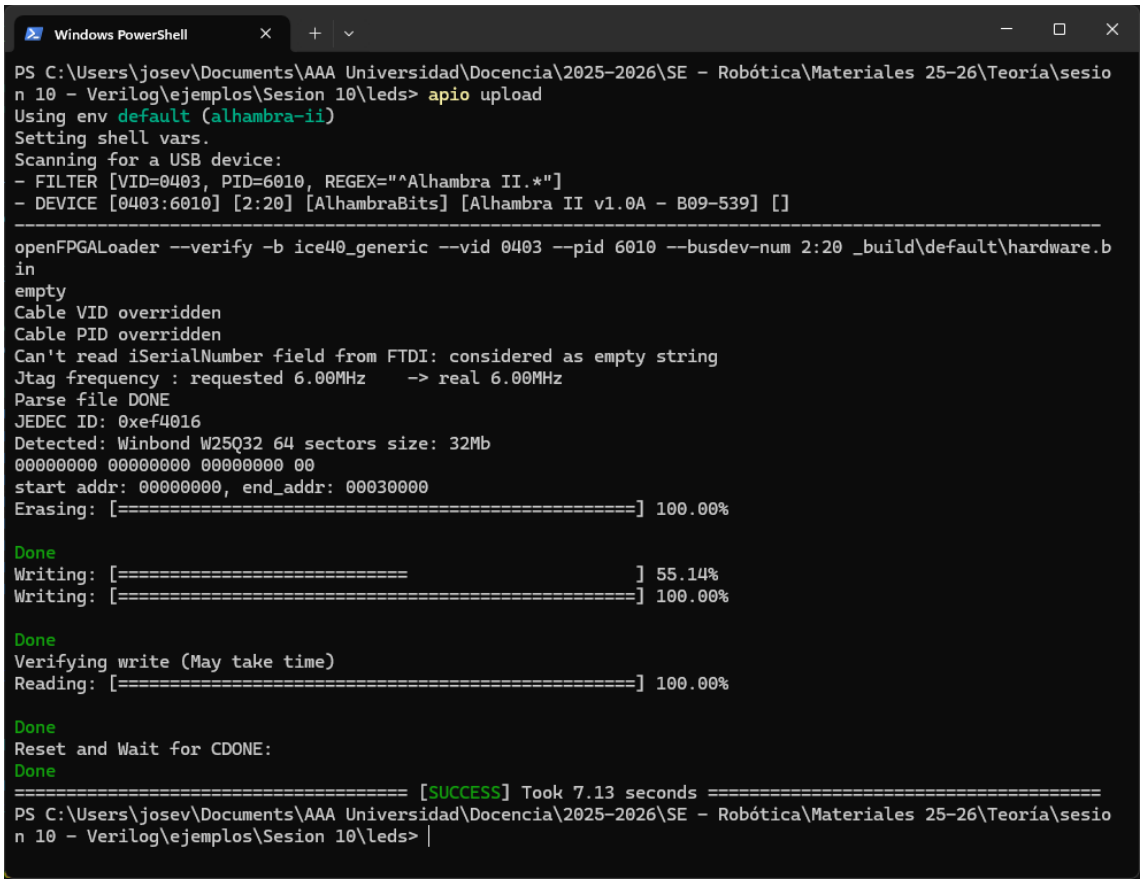
Ahora vamos a sintetizar este código para la placa FPGA. Para ello es necesario añadir un cuarto archivo en el cual se indica el esquema de conexión de pines entre el código verilog y la placa. Añade un archivo llamado `leds.pcf` con este contenido:

```
set_io  LED0  45
set_io  LED1  44
set_io  LED2  43
set_io  LED3  42
set_io  LED4  41
set_io  LED5  39
set_io  LED6  38
set_io  LED7  37
```

Para sintetizar el hardware, ejecuta `apio build`.



Y finalmente carga el sintetizado en la plaza: `apio upload`



Nota: los mensajes de salida de los comandos pueden ser diferentes dependiendo de la versión de APIO instalada.

Ejemplo 3 – Leds parpadeo

En este ejemplo vamos a implementar también un hardware que utilice leds, pero realizará el parpadeo valiéndose de la señal de reloj de la placa ICEZUM/ALHALMBRA II. El código del ejemplo `leds_blink.v` es el siguiente:

```
module LEDS_blink (
    input  CLK,   // 12MHz clock
    output LED7,  // LED to blink
    // The rest of the LEDs are turned off.
    output LED6, LED5, LED4,  LED3, LED2, LED1, LED0
);

  reg [23:0] counter = 0;

  // para usar con la simulación
  always @(posedge CLK) counter[23] <= ~counter[23];

  // para usar con la el reloj de 12MHz
  //always @(posedge CLK) counter <= counter + 1;

  assign LED7 = counter[23];

  //-- Turn off the other LEDs
  assign {LED6, LED5, LED4, LED3, LED2, LED1, LED0} = 3'b0;

endmodule
```

Este código simple declara una entrada (CLK) y como salidas todos los LEDs de la placa. Crea un registro que hará de contador, y en cada flanco alto del reloj va a realizar una acción. Hemos dejado dos posibilidades, una para simulación y otra para cuando se utilice la FPGA que tiene un reloj de 12MHz. El parpadeo se produce cada vez que cambia el bit 23, es decir, cada 8388608 ciclos (con la frecuencia de 12Mhz de Alhambra II, el cambio sería cada 0.6999 seg).

Esta opción: `always @(posedge CLK) counter[23] <= ~counter[23];`. Se utilizará junto a la simulación, ya que como se verá se ha preparado el testbench para que la señal de reloj cambie cada segundo.

Esta opción: `always @(posedge CLK) counter <= counter + 1;` . Se utilizará cuando se despliegue el sintetizado en la placa, ya que la FPGA con su reloj a 12MHz implicará que, al incrementar en 1 el contador de 24 bits, el bit 23 cambiará aproximadamente cada 12MHz, es decir, una vez por segundo.

Además del archivo verilog, es necesario un `apio.ini`:

```
[env:default]
board = alhambra-ii
top-module = leds_blink
default-testbench = leds_blink_tb.v
```

El archive de testbench `leds_blink_tb.v`, que contendrá lo siguiente:

```
`timescale 1ms/100us

module LEDS_blink_tb;
    reg CLK;
    wire LED7, LED6, LED5, LED4, LED3, LED2, LED1, LED0;
    parameter CLK_PERIOD = 500;
    parameter SIM_TIME = 10000;

    LEDS_blink uut (
     .CLK(CLK),
     .LED7(LED7), .LED6(LED6), .LED5(LED5), .LED4(LED4),
     .LED3(LED3), .LED2(LED2), .LED1(LED1), .LED0(LED0)
    );

    initial begin
        CLK = 0;
    end

    always begin
        #(CLK_PERIOD) CLK = ~CLK;
    end

    initial begin
        $dumpvars(0, LEDS_blink_tb);

        $display("--- Simulacion de LED_blink Iniciada ---");
         $display("Tiempo(ns) | CLK | LED7 | LED6 LED5 LED4 LED3
LED2 LED1 LED0");
        $monitor("%5t      | %b   | %b | %b %b %b %b %b %b %b",
$time, CLK, LED7, LED6, LED5, LED4, LED3, LED2, LED1, LED0);

        #(SIM_TIME);

       $display("--- Simulacion Finalizada en %t ns ---", $time);
        $finish;
    end
endmodule
```

El testbench tiene una escala temporal de 1ms, con un aprecisión de 100ms. Dado que deseamos ver parpadear el led cada 1 seg, con esta escala es suficiente. Se declara como reg las entradas (CLK) y como wire el resto de señaes.

Se declaran dos parámetros, CLK_PERIOD nos servirá para indicar cada cuanto cambia de flanco el reloj. SIM_PERIOD define el tiempo total de simulación.

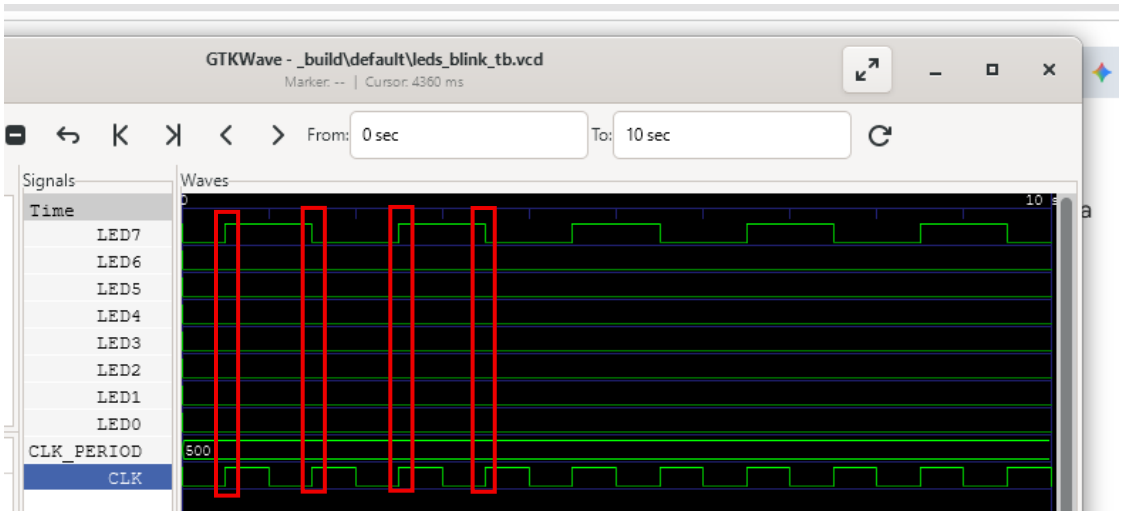
Los bloques initial solo se ejecutarán una vez. El primer bloque inicializa el registro CLK a 0.

El bloque always es un bloque que se va a repetir durante toda la simulación de forma concurrente. Dentro de este bloque se realiza un retraso temporal de

CLK_PERIOD, es decir, de 500ms (ya que es 500 veces la escala temporal de 1ms definida en timescale). O lo que es lo mismo, 0,5 seg. Y lo que hace tras el retraso es invertir el valor de CLK, de forma que CLK va alternando entre 0 y 1 cada 0,5 segundos.

El último bloque initial se encarga de monitorizar todas las variables para poder ver la simulación.

Si ejecutamos la simulación veremos algo como esto:



Como se puede ver, en los flancos de subid a de CLK es cuando se invierte la señal del LED7, que es el comportamiento esperado.

Ahora podemos sintetizar el hardware. Para eso vamos a necesitar indicar los pines a los que se conecta de nuestra placa utilizando `leds_blink.pcf`, en una Alhambra II sería el siguiente:

```
# Configuracion del Reloj principal (12 MHz)
set_io CLK 49
# Configuracion del LED de Usuario
set_io LED7 45
set_io LED6 44
set_io LED5 43
set_io LED4 42
set_io LED3 41
set_io LED2 39
set_io LED1 38
set_io LED0 37
```

Como se puede ver se conectan las variables LED a cada pin físico de la placa, y la señal CLK al pin de reloj de la placa.

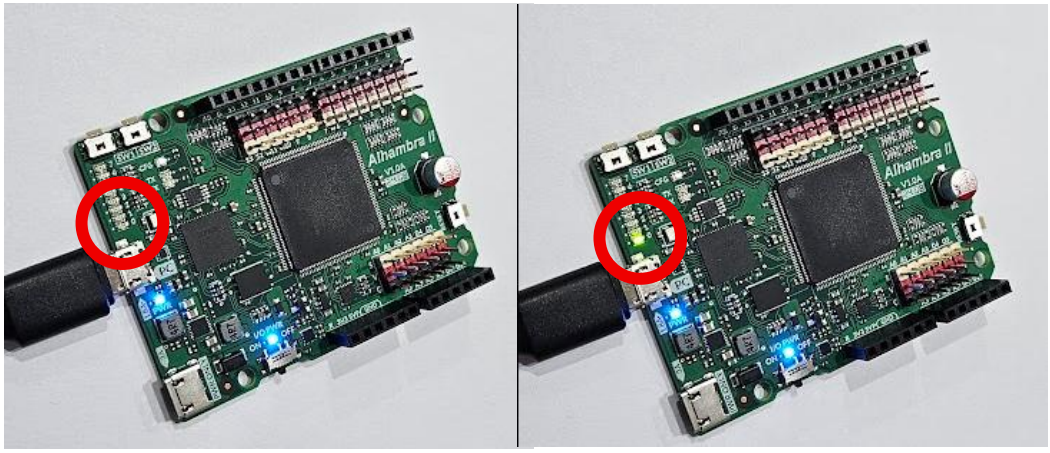
Finalmente, en el código leds_blink.v descomentamos está línea:

```
always @(posedge CLK) counter <= counter + 1;
```

Y comentamos otra línea always. Ya podemos hacer el build y posteriormente el upload.

```
Windows PowerShell
PS C:\Users\josev\Documents\AAA Universidad\Docencia\2025-2026\SE - Robótica\Materiales 25-26\Teoría\sesion 10 - Verilog
\ejemplos\Sesion 10\leds_blink> apio build
Using env default (alhambra-ii)
Setting shell vars.
------------------------------------------------------------------------------------------------------------------------
yosys -p "synth_ice40 -top leds_blink -json _build\default\hardware.json" -q leds_blink.v
scons: `build' is up to date.
========================================== [SUCCESS] Took 5.68 seconds ==========================================
PS C:\Users\josev\Documents\AAA Universidad\Docencia\2025-2026\SE - Robótica\Materiales 25-26\Teoría\sesion 10 - Verilog
\ejemplos\Sesion 10\leds_blink> apio upload
Using env default (alhambra-ii)
Setting shell vars.
Scanning for a USB device:
- FILTER [VID=0403, PID=6010, REGEX="^Alhambra II.*"]
- DEVICE [0403:6010] [2:11] [AlhambraBits] [Alhambra II v1.0A - B09-539] []
------------------------------------------------------------------------------------------------------------------------
openFPGALoader --verify -b ice40_generic --vid 0403 --pid 6010 --busdev-num 2:11 _build\default\hardware.bin
empty
Cable VID overridden
Cable PID overridden
Can't read iSerialNumber field from FTDI: considered as empty string
Jtag frequency : requested 6.00MHz    -> real 6.00MHz
Parse file DONE
JEDEC ID: 0xef4016
Detected: Winbond W25Q32 64 sectors size: 32Mb
00000000 00000000 00000000 00
start addr: 00000000, end_addr: 00030000
Erasing: [==================================================] 100.00%

Done
Writing: [============================                      ] 54.57%
Writing: [==================================================] 100.00%

Done
Verifying write (May take time)
Reading: [==================================================] 100.00%

Done
Reset and Wait for CDONE:
Done
========================================== [SUCCESS] Took 6.95 seconds ==========================================
PS C:\Users\josev\Documents\AAA Universidad\Docencia\2025-2026\SE - Robótica\Materiales 25-26\Teoría\sesion 10 - Verilog
\ejemplos\Sesion 10\leds_blink>
```

Ahora si miramos la placa, veremos parpadear el led más cercano al conector USB mientras el resto permanecen apagados.



Ejemplo 3b – Leds parpadeo 1 seg

Este ejemplo es similar al anterior, pero el cambio del estado del bit en vez de estar ligado a un contador, va a estar ligado a tiempo. Se realiza cada 1 segundo. Además hemos añadido un botón para poder hacer reset.

```
module leds_blink (
    input wire clk,
    input wire rst,
    output reg led
);

    parameter FREQ_CLOCK = 12_000_000;
    parameter CYCLES_TO_TOGGLE = FREQ_CLOCK - 1;


    reg [23:0] counter;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0;
            led <= 0;
        end else begin
            if (counter == CYCLES_TO_TOGGLE) begin
                led <= ~led;
                counter <= 0;
            end else begin
                counter <= counter + 1;
            end
        end
    end

endmodule
```

Como se puede ver, la estrategia en este caso es detectar cuando se cuentan un cierto número de ciclos de reloj, que coinciden con los mismos que incluye 1 segundo a una velocidad determinada. Para Alhambra II con un reloj de 12MHz, eso implica 12.000.000 ciclos.

Ejemplo 4 – Botón a led

Ahora en el siguiente ejemplo vamos a hacer uso de un botón de la placa, de forma que hagamos que al pulsarlo se active o desactive un led directamente. El botón hará las funciones de interruptor, de forma que al pulsarlo se active el led, y al soltarlo se apague. Esto es una lógica combinacional simple y genera el siguiente código `button_led_direct.v`:

```
module button_led_direct (
    input wire BTN,
    output wire LED
);
    assign LED = BTN;

endmodule
```

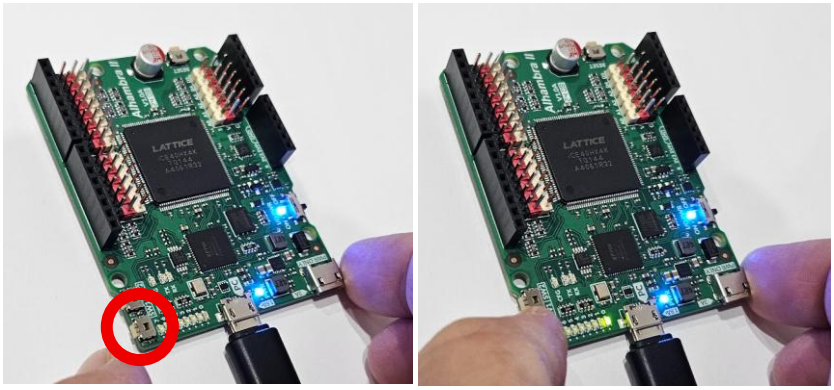
Como se puede ver, el código simplemente conecta el botón con el led, de forma que cuando se pulse, el led recibirá la señal del botón, y cuando se suelte, el led recibirá 0. Los botones de la Alhambra II se activan en alto, es decir, cuando se pulsa emite 1, cuando no se pulsa emite 0. Para poder sintetizar el hardware, utilizamos el siguiente código `button_led_direct.pcf`:

```
# Este botón está "Activo en Alto" (pulsa = 1, no pulsa = 0)
set_io BTN 34
# Uno de los LEDs de usuario de la placa
set_io LED 45
```

Simplemente mirando el pinout de la placa, podemos ver los pines a los que se corresponde cada señal. Ahora se puede hacer el build y el upload.

```
Windows PowerShell
PS C:\Users\josev\Documents\AAA Universidad\Docencia\2025-2026\SE - Robótica\Materiales 25-26\Teoría\sesion 10 - Verilog
\ejemplos\Sesion 10\button_led_direct> apio build
Using env default (alhambra-ii)
Setting shell vars.
------------------------------------------------------------------------------------------------------------------------
yosys -p "synth_ice40 -top button_led_direct -json _build\default\hardware.json" -q button_led_direct.v
nextpnr-ice40 --hx8k --package tq144:4k --json _build\default\hardware.json --asc _build\default\hardware.asc --report _
build\default\hardware.pnr --pcf button_led_direct.pcf -q
icepack _build\default\hardware.asc _build\default\hardware.bin
========================================== [SUCCESS] Took 5.56 seconds ==========================================
PS C:\Users\josev\Documents\AAA Universidad\Docencia\2025-2026\SE - Robótica\Materiales 25-26\Teoría\sesion 10 - Verilog
\ejemplos\Sesion 10\button_led_direct> apio upload
Using env default (alhambra-ii)
Setting shell vars.
Scanning for a USB device:
- FILTER [VID=0403, PID=6010, REGEX="^Alhambra II.*"]
- DEVICE [0403:6010] [2:10] [AlhambraBits] [Alhambra II v1.0A - B09-539] []
------------------------------------------------------------------------------------------------------------------------
openFPGALoader --verify -b ice40_generic --vid 0403 --pid 6010 --busdev-num 2:10 _build\default\hardware.bin
empty
Cable VID overridden
Cable PID overridden
Can't read iSerialNumber field from FTDI: considered as empty string
Jtag frequency : requested 6.00MHz    -> real 6.00MHz
Parse file DONE
JEDEC ID: 0xef4016
Detected: Winbond W25Q32 64 sectors size: 32Mb
00000000 00000000 00000000 00
start addr: 00000000, end_addr: 00030000
Erasing: [==================================================] 100.00%

Done
Writing: [============================                      ] 54.38%
Writing: [==================================================] 100.00%

Done
Verifying write (May take time)
Reading: [==================================================] 100.00%

Done
Reset and Wait for CDONE:
Done
========================================== [SUCCESS] Took 6.71 seconds ==========================================
```

Y en la placa, podemos ver el comportamiento al pulsar el botón SW1.

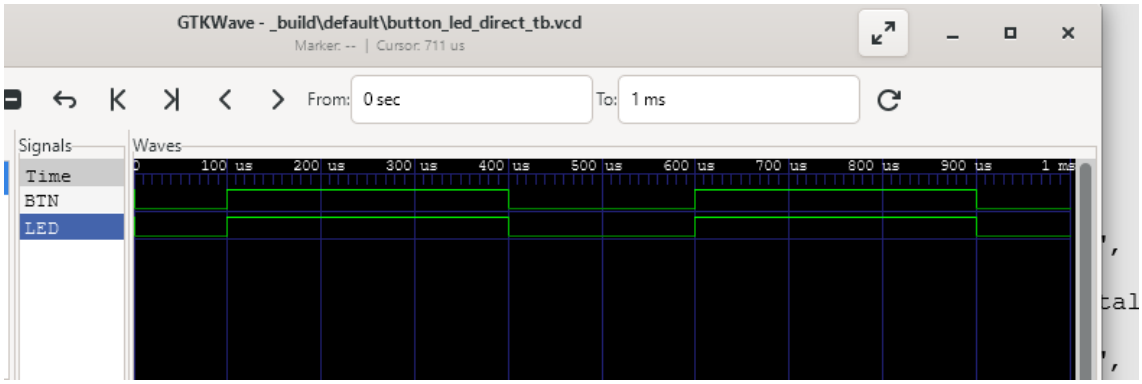


También podemos generar el testbench, y probar que efectivamente el botón controla el led. Para eso, el código de `button_led_direct_tb.v` sería el siguiente:

```
`timescale 1ns / 1ps

module button_led_direct_tb;

    reg BTN;
    wire LED;

    button_led_direct uut (
        .BTN(BTN),
        .LED(LED)
```

```
    );

    initial begin

   $dumpvars(0, button_led_direct_tb);

   $display("Iniciando simulación...");
   BTN = 0; // Botón suelto inicialmente

   #100000; // Esperar 100 us
   BTN = 1; // Presionar botón
   $display("Tiempo: %0t | BTN = %b | LED = %b", $time, BTN, LED);

   #300000; // Mantener presionado 300 us (Total: 400 us)
   BTN = 0; // Soltar botón
   $display("Tiempo: %0t | BTN = %b | LED = %b", $time, BTN, LED);

   #200000; // Esperar 200 us (Total: 600 us)
   BTN = 1; // Presionar de nuevo
   $display("Tiempo: %0t | BTN = %b | LED = %b", $time, BTN, LED);

   #300000; // Mantener presionado 300 us (Total: 900 us)
   BTN = 0; // Soltar
   $display("Tiempo: %0t | BTN = %b | LED = %b", $time, BTN, LED);

   // Completar el tiempo restante para llegar a 1ms
   #100000; // (Total: 1,000,000 ns = 1 ms)

   $display("Fin de la simulación.");
   $finish; // Detener la simulación

    end
endmodule
```

Al simular con apio sim este testbench, podremos ver algo como esto: el botón se pulsa a los 100 us, se suelta 300 us después, se vuelve a pulsar 200 us y se suelta 300us.



Ejemplo 5 – Botón a led con memoria

En el siguiente ejemplo vamos a hacer uso de los botones de una forma más sofisticada. Ahora el botón va a tener memoria, de forma que cuando se pulse el led se encienda, y cuando se vuelva a pulsar, el led se apague. En el ejemplo anterior, al no haber memoria, se podía utilizar un circuito combinacional. Ahora, al tener memoria, esto obliga a mantener registros y por tanto a utilizar la señal de reloj, generando un circuito secuencial. El funcionamiento va a ser simple, al pulsar un botón un led se enciende, y si está encendido se apaga. El código de `button_led_toggle.v` va a ser el siguiente:

```
module button_led_toggle (
    input wire CLK,
    input wire BTN,
    output reg LED
);

reg btn_current = 1'b0;
reg btn_last = 1'b0;

reg btn_pressed = 1'b0;

always @(posedge CLK) begin
    btn_last <= btn_current;
    btn_current <= BTN;

    if (btn_last == 1'b0 && btn_current == 1'b1) begin
        btn_pressed <= 1'b1; // Señal de pulsación
    end else begin
        btn_pressed <= 1'b0; // No hay pulsación
    end
end

always @(posedge CLK) begin
    if (btn_pressed) begin
        // Cambia el estado del LED (de 0 a 1 o de 1 a 0)
        LED <= ~LED;
    end
end

initial begin
    LED = 1'b0;
end

endmodule
```

Vamos a analizar la lógica del programa.

En la declaración del módulo, se declaran 2 señales de entrada: CLK y BTN. Estas señales estarán unidas al reloj de la placa y a un botón. Se declara también una salida que en este caso es de tipo registro, LED. Es necesario que sea un registro

pues el LED irá alternando entre encendido y apagado cuando se pulse y suelte el botón, por tanto en algún lugar del código y bajo alguna condición se producirá la alternancia del estado del LED, algo así como `LED<=~LED`. Se utiliza la asignación no bloqueante pues LED es un registro.

Para generar el cambio, este código se va a basar en detectar cuando el botón pasa de estar no pulsado a pulsado. Es decir, va a detectar el flanco de activación, el momento en el que pasa de 0 a 1. Debemos tener en cuenta que al tener ahora un circuito secuencial, en cada ciclo de reloj, podemos hacer comprobaciones tales como si ha cambiado el estado del botón, y que cuando ese cambio se produzca entonces cambiar el valor de LED. La siguiente imagen muestra un ejemplo de cómo el LED cambia de estando cuando se produce el flanco de subida en la activación. Estas comprobaciones ocurrirán durante los ciclos de reloj.

```
CLK _|¯|_|¯|_|¯|_|¯|_|¯|

LED __|¯¯¯¯¯|_____|¯¯¯¯¯

BTN __|¯¯|__|¯¯|__|¯¯|__
      ↑     ↑     ↑
      flanco positivo
```

Cómo lo queremos detectar es el cambio de 0→1 del botón, vamos a almacenar en dos variables el estado del botón anterior y el estado del botón actual, estas variables las vamos a inicializar a 0.

```
reg btn_current = 1'b0;
reg btn_last = 1'b0;
```

En el primer always, en cada flanco de subid a del reloj se hace una asignación a los registros de memoria que almacenan el estado anterior y el estado actual:

```
always @(posedge CLK) begin
    btn_last <= btn_current;
    btn_current <= BTN;
    ...
```

Como se puede ver, en el estado anterior se almacena el valor de current, y en current el estado del botón. De esta forma siempre tenemos en las variables last y current el estado anterior y el actual del botón. En la siguiente condición simplemente comprobamos si se ha producido un cambio de 0 a 1:

```
if (btn_last == 1'b0 && btn_current == 1'b1) begin
        btn_pressed <= 1'b1; // Señal de pulsación
    end else begin
        btn_pressed <= 1'b0; // No hay pulsación
    end
```

Al utilizase asignación no bloqueante, hay que tener en cuenta que ocurrirán todas a la vez, es decir, LAST tomará el valor de current, en current se signará el valor de BTN, y en btn_pressed se resolvera a 1 o 0 en función de la condidición. Por tanto, solo en el ciclo de reloj, donde se detecta este cambio el valor de btn_pressed tendrá asignado el valor de 1.

Finalmente el ultimo bloque always lo que hace en cada flanco de subida es comprobar si en btn_pressed hay un 1. En tal caso, invierte el valor de LED.

```
if (btn_pressed) begin
    LED <= ~LED;
end
```

Como esto solo puede darse durante un ciclo de reloj, al pulsar el botón se encenderá o apagará el LED correspondiente.

Para sintetizar este hardware, debemos crear el archivo button_led_toggle.pcf, que es muy sencillo, pues solo debemos unir las líneas de reloj y el botón, y el led de salida.

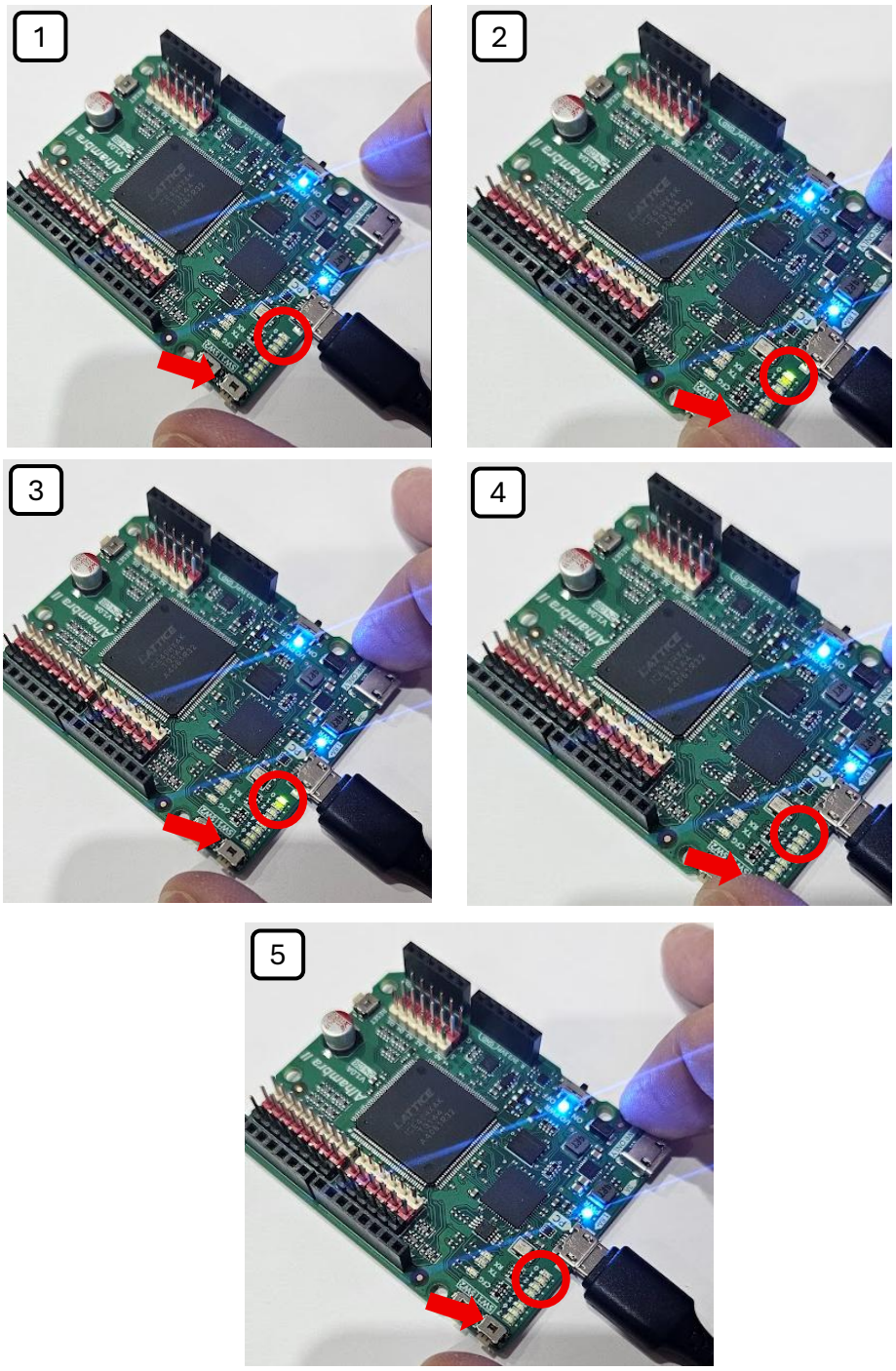
```
set_io CLK 49
set_io BTN 34
set_io LED 45
```

Finamente para compilar, creamos el `apio.ini` con este contenido:

```
[env:default]
board = alhambra-ii
top-module = button_led_toggle
```

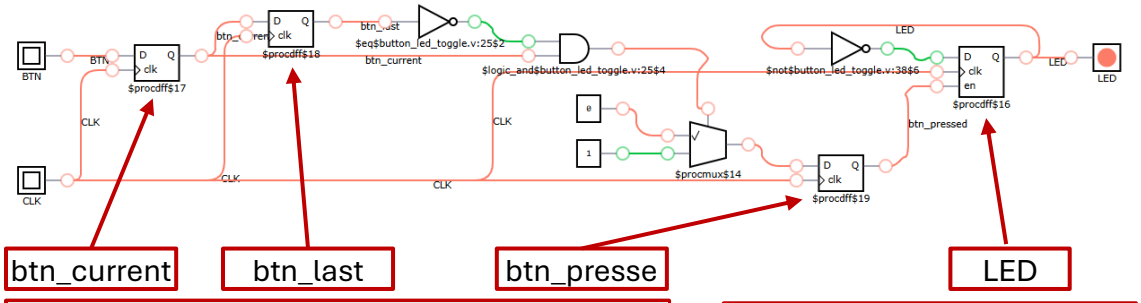
Ahora Podemos hacer el build y el upload. Y cuando carguemos el sintetizado en nuestra placa, podremos ver lo siguiente:

- 1: se ve como en la placa está apagado el led, antes de pulsar el botón.
- 2: se muestra como en el momento que se pulsa el botón, antes de haberlo soltarlo, el led ya se ha encendido.
- 3: soltamos el botón y el led permanece encendido.
- 4: volvemos a pulsar el botón, y el led se apaga en ese instante
- 5: soltamos el botón, y el led permanece apagado.



Si queremos echar un vistazo a como es este sintetizado en hardware, podemos utilizar la herramienta DigitalJS, https://digitaljs.tilk.eu/

En esta herramienta online, podemos cargar un código de verilog y nos mostrará en elementos básicos el circuito. Si cargamos nuestro archivo button_led_toggle.v, y pulsamos el botón RUN, podemos ver el circuito siguiente:



btn_current

btn_last

btn_presse

LED

```
always @(posedge CLK) begin
    btn_last <= btn_current;
    btn_current <= BTN;
    if (btn_last == 1'b0
                && btn_current == 1'b1) begin
        btn_pressed <= 1'b1;
    end else begin
        btn_pressed <= 1'b0;
    end
```

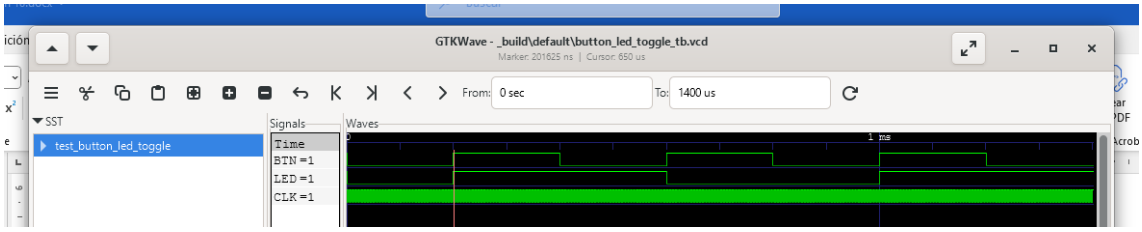
```
always @(posedge CLK) begin
    if (btn_pressed) begin
        LED <= ~LED;
    end
end
```

Por último, también es interesante hacer la simulación de este circuito porque nos puede aportar claridad en cómo es la ejecución. Para eso necesitamos el archivo button_led_toggle_tb.v siguiente:

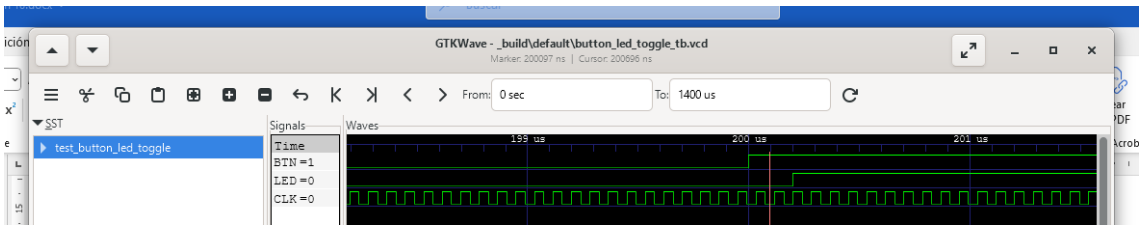
```
`timescale 1ns / 1ps
module test_button_led_toggle;
    reg CLK;     // Reloj
    reg BTN;     // Botón
    wire LED;    // LED (salida del módulo DUT)


    button_led_toggle dut (
        .CLK(CLK),
        .BTN(BTN),
        .LED(LED)
    );


    parameter CLK_PERIOD = 83.33; // 12 MHz en ns

    always begin
        CLK = 1'b0;
        #(CLK_PERIOD / 2);
        CLK = 1'b1;
        #(CLK_PERIOD / 2);

    initial begin
        // 1. Inicialización
        BTN = 1'b0; // Botón no presionado
        CLK = 1'b0; // Inicializa el reloj

        $dumpvars(0, test_button_led_toggle);

        $display("Inicio de la simulacion...");
        $display("Tiempo | CLK | BTN | LED");
        $monitor("%0t | %b   | %b   | %b", $time, CLK, BTN, LED);
```

```
        #200000; // Espera un poco

        // --- Primera Pulsación de Botón ---
        $display("\n--- Primera pulsacion ---");
        BTN = 1'b1; // Presionar botón
        #200000; // Mantener presionado
        BTN = 1'b0; // Soltar botón
        #200000; // Esperar

        // --- Segunda Pulsación de Botón ---
        $display("\n--- Segunda pulsacion ---");
        BTN = 1'b1; // Presionar botón
        #200000; // Mantener presionado
        BTN = 1'b0; // Soltar botón
        #200000; // Esperar

        // --- Tercera Pulsación de Botón ---
        $display("\n--- Tercera pulsacion ---");
        BTN = 1'b1; // Presionar botón
        #200000; // Mantener presionado
        BTN = 1'b0; // Soltar botón
        #200000; // Esperar

        // --- Finalizar Simulación ---
        $display("\nFin de la simulacion.");
        $finish; // Termina la simulación
    end

endmodule
```

Al ejecutar la simulación veremos la siguiente figura de onda.



Si hacemos zoom justo en una zona donde se simula la pulsación del botón podemos ver esto:



Ejemplo 6 – Botón con pulsación larga

En este ejercicio vamos a proponer continuar con el uso de un botón que activa o desactiva un led, pero en este caso en lugar de utilizar un flanco de subida, es decir, el instante donde se pulsa el botón, va a ser necesario que el botón permanezca pulsado durante 1 seg. para activar el led.

Para ello comenzaremos con el programa verilog. Si pensamos en cómo hacer esta funcionamiento a nivel de hardware (hay que pensar distinto a cuando se hace en software), debemos tener en cuenta que vamos a necesitar utilizar el reloj del sistema. Como estamos utilizando un Alhambra II/iCEZUM, podemos considerar un segundo equivale a 12.000.000 de ciclos de reloj (12MHz), por lo que nuestro código puede ser algo similar a esto:

```
module long_press_toggle (
    input wire CLK, // Reloj de la FPGA (12 MHz)
    input wire BTN, // Botón, Activo en Alto (1 = Presionado)
    output reg LED // Salida del LED (memoria del estado)
);

    parameter CLK_FREQ = 12_000_000;
    parameter ONE_SECOND = CLK_FREQ;
    parameter COUNT_BITS = 24;
    reg [COUNT_BITS-1:0] counter = 0;

    initial begin
        LED = 1'b0;
    end

    always @(posedge CLK) begin
        if (BTN == 1'b1) begin
            if (counter < ONE_SECOND) begin
                counter <= counter + 1;
            end else if (counter == ONE_SECOND) begin
                LED <= ~LED;
                counter <= counter + 1;
            end else begin
                counter <= counter;
            end
        end else begin
            counter <= 0;
        end
    end
endmodule
```

En este código se utiliza un registro de 24 bits (COUNT_BITS-1) que nos va a permitir contar hasta 12.000.000+1. Si te fijas en el código lo que se hace es lo siguiente. En cada flanco de subida del reloj, se determina si el botón está pulsado. En tal caso, se comprueba si counter es inferior a ONE_SECOND (que tiene como valor el número de ciclos por segundo del reloj). Si es inferior, se incrementa en 1. En caso

de que counter haya alcanzado el valor de ONE_SECOND, entonces se invierte el valor de LED (apagado pasa a encendido y viceversa), y se incrementa counter en 1. En caso contrario que haya superado ONE_SECOND entonces no hace nada pues a counter le asigna el valor de counter. El último else cubre el caso en que el botón no está pulsado, ante lo cual establece counter a 0.

Si queremos sintetizar este hardware, podemos crear el archivo `logn_press_toggle.pcf` con el contenido:

```
set_io CLK 49
set_io BTN 34
set_io LED 45
```

Esto conecta el reloj a la señal CLK, el botón al primer botón de la plaza y el LED al primer led. El resultado sobre nuestra placa es muy similar al ejemplo anterior, pero ahora es necesario pulsar durante 1 segundo para que se active.

Realizar el testsbench de este código puede ser un problema ya que en realidad nuestro hardware necesita mantener el botón pulsado durante 12 millones de ciclos para incrementar el contador lo suficiente hasta que se active el led. Esto significa que si almacenamos los valores de las variables cuando cambian, será necesario almacenar varios millones de ciclos de reloj. Por tanto el archivo de salida donde se almacena la simulación (`long_press_toggle_tb.vcd`) va a ocupar decenas de megabytes. El testbench que generaría una simulación conforme a la declaración de hardware que tenemos sería el siguiente `long_press_toggle_tb.v`:

```
`timescale 1ns / 1ps
module long_press_toggle_tb;
    reg CLK;
    reg BTN;
    wire LED;
    long_press_toggle uut (
        .CLK(CLK),
        .BTN(BTN),
        .LED(LED)
    );
    initial CLK = 0;
    always #41.667 CLK = ~CLK;  // genera un CLK aprox de 12MHz

    initial begin
        $dumpvars(0, long_press_toggle_tb);
        BTN = 0; // Botón suelto
          $display("Incio: Tiempo %0d ms | BTN=%b | LED=%b",
$time/1000000, BTN, LED);
        #100_000_000;
        // --- PRUEBA 1: Pulsación CORTA (0.2s) ---
        $display("--- Pulsación Corta (0.2s) ---");
        BTN = 1; // Presionar
        #200_000_000; // Esperar 0.2s
```

```
        BTN = 0; // Soltar
          $display("Fin Corta: Tiempo %0d ms | BTN=%b | LED=%b
(Esperado: 0)", $time/1000000, BTN, LED);

        // Esperamos un hueco de 0.2 segundos
        #200_000_000;

        // --- PRUEBA 2: Pulsación LARGA (> 1.0s) ---
        $display("--- Pulsación Larga (Inicio) ---");
        BTN = 1;

        #1_200_000_000; // Mantenemos presionado 1.2 segundos

        BTN = 0; // Soltar (Tiempo aprox: 1.7s de simulación)
          $display("Fin Larga: Tiempo %0d ms | BTN=%b | LED=%b
(Esperado: 1)", $time/1000000, BTN, LED);

        #300_000_000;

        $display("Fin de simulación a los 2 segundos.");
        $finish;
    end

endmodule
```

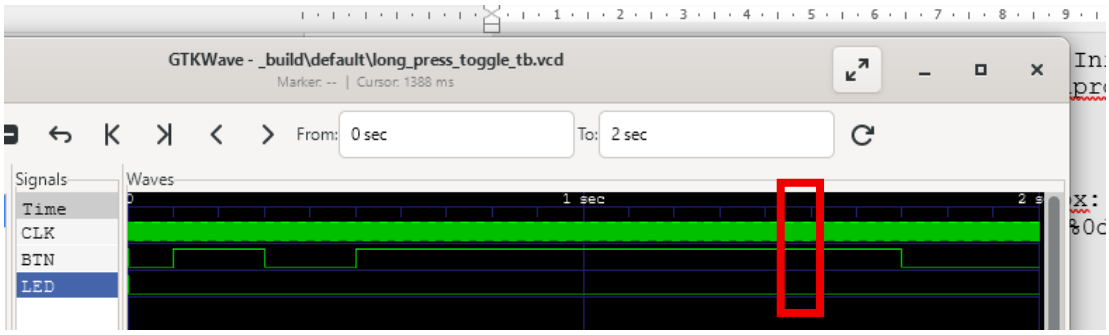
Si tratas de ejecutar la simulación con este testbench veras que tarda mucho y genera un archivo inmenso. Entonces podemos probar a trucar la simulación, por ejemplo, pensando que nuestro reloj, en vez de operar a 12 MHz, puede operar a 1 KHz, es decir, que el reloj tendrá 1000 ciclos en un segundo. Para eso modificaremos un poco el archivo de testbench, dejándolo de la siguiente forma (fíjate en las partes marcadas en rojo que son los cambios).

```
`timescale 1ms / 100us

module long_press_toggle_tb;

    reg CLK;
    reg BTN;
    wire LED;

    long_press_toggle uut (
        .CLK(CLK),
        .BTN(BTN),
        .LED(LED)
    );


    initial CLK = 0;

    always #0.5 CLK = ~CLK;

    initial begin
        $dumpvars(0, long_press_toggle_tb);
```

```
        BTN = 0; // Botón suelto
       $display("Incio: Tiempo %0d ms | BTN=%b | LED=%b", $time/1,
BTN, LED);

        #100;

        $display("--- Pulsación Corta (0.2s) ---");
        BTN = 1;
        #200;
        BTN = 0;
          $display("Fin Corta: Tiempo %0d ms | BTN=%b | LED=%b
(Esperado: 0)", $time/1, BTN, LED);

        #200;


        $display("--- Pulsación Larga (Inicio) ---");
        BTN = 1; // Presionar (Tiempo aprox: 0.5s de simulación)

        #1_200;

        BTN = 0; // Soltar (Tiempo aprox: 1.7s de simulación)
          $display("Fin Larga: Tiempo %0d ms | BTN=%b | LED=%b
(Esperado: 1)", $time/1, BTN, LED);

        #300;

        $display("Fin de simulación a los 2 segundos.");
        $finish;
    end

endmodule
```

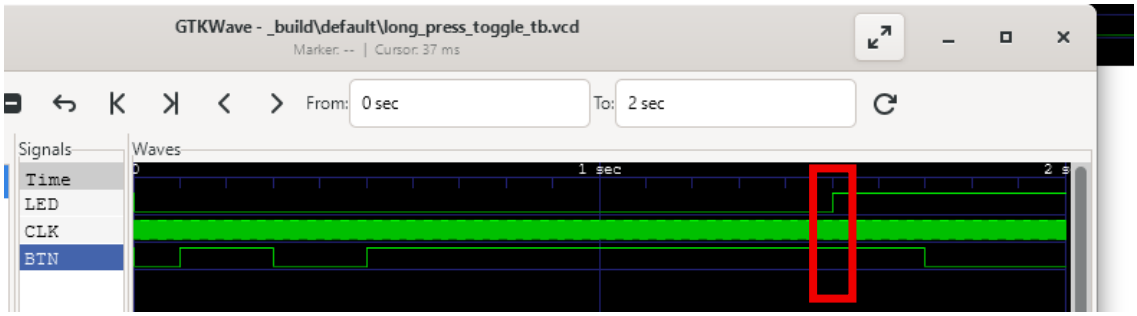
Ahora la escala temporal es de 1ms, y si te fijas el reloj cambia cada `#0.5` del tiempo de la escala temporal, por tanto, en 1 ms contendrá el flanco de subida y de bajada. El resto de los cambios simplemente cambia la escala dividiendo los valores por 10^6 que es lo que se ha reducido la frecuencia (de MHz a KHz).

Por último, si intentas simular con este testbench verás que la onda de salida no se corresponde con el resultado esperado, ya que verá salgo similar a esto:



Como puedes observar, donde está la marca roja debería haberse encendido el led, ya que el botón ha estado pulsado durante 1 segundo. ¿Por qué no se enciende el LED?

Esto es debido a que hemos truncado el testbench, de forma que ahora en lugar de a una frecuencia de reloj de 12MHz utilizar 1KHz, y por tanto, en nuestro código de verilog también debemos cambiar el parámetro CLK_FREQ de esta forma: `parameter CLK_FREQ = 1_000;` ya que ahora el contador no debe llegar hasta los 12.000.000 sino hasta 1000. Si cambiamos este parámetro de esta forma, veremos que la simulación si arroja los resultados esperados.



Ejemplo 7 – Luces Kitt

Este ejercicio es muy vistoso, y como puedes suponer, se trata de simular las luces del famoso coche fantástico utilizando los leds de tu tarjeta. Vamos a pensar en la lógica de este hardware:

- De los 8 leds, solo habrá encendido 1. Se encenderán de derecha a izquierda o de izquierda a derecha. Es decir, es necesario controlar el sentido de encendido.
- Deberá transcurrir un tiempo entre el encendido de uno y el siguiente, por lo que hay una espera.
- Inicialmente estará encendido el primer led.

El código que hace esta lógica es el siguiente `kitt_lights.v`, vamos a analizarlo por partes:

```
module kitt_lights (
    input wire CLK,          // Reloj de la FPGA (12 MHz)
    output reg [7:0] LED_OUT // Salida para los 8 LEDs
);
```

La declaración del módulo, vamos a usar la línea de reloj y un registro de 8 bits, un bit que estará unido a los leds.

```
parameter CLK_FREQ = 12_000_000;
parameter DELAY_MS = 200;
parameter DELAY_CYCLES = (CLK_FREQ / 1000) * DELAY_MS;
```

Después declaramos 3 parámetros. El primero define la frecuencia de reloj de nuestra placa (12MHz), el segundo indica en milisegundos el tiempo de retardo entre el encendido de cada led, y el tercero convierte el retardo en milisegundos en ciclos de reloj.

```
parameter COUNT_BITS = 22;
reg [COUNT_BITS-1:0] delay_counter = 0;
reg delay_done = 1'b0;
```

Esta sección declara un contador de bists, crear un registro de 22 bits denominado delay_counter (que será utilizado para contar el retardo entre led y led) y un registro de 1 bit denominado delay_done, que será utilizado para indicar cuando se ha cumplido el delay entre leds. Es muy importante que para contar tiempo en una FPGA debemos contar **CILOS DE RELOJ**. Es por tanto que si queremos esperar por ejemplo 1 segundo, debemos contar en realidad 12.000.000 de ciclos en una placa a 12MHz.

```
reg [2:0] current_led_index = 3'b000;
reg direction = 1'b0; // 0 = Der (0 -> 7), 1 = Izq (7 -> 0)
```

Finalmente, un índice para saber qué led hay que cambiar en tras cada espera, y un bit direction que nos indica en qué dirección se están moviendo los leds.

```
always @(posedge CLK) begin
  if (delay_counter == DELAY_CYCLES - 1) begin
    delay_counter <= 0;
    delay_done <= 1'b1;
  end else begin
    delay_counter <= delay_counter + 1;
    delay_done <= 1'b0;
  end
end
```

Este bloque always se ejecuta en cada flanco de subida de reloj. Si ves la lógica comprueba si el registro ha alcanzado el valor de DELAY_CYCLES-1. Cuando alcanza este valor, cambia el valor de delay_counter a 0 y el de dealy_done a 1. Utiliza asignaciones no bloqueantes. En caso de que no se haya llegado a este valor, se incrementa el delay_counter y se pone a 0 delay_done. Es decir, este bloque activará delay_done cuando se hayan llegado a los DELAY_CYCLES-1 contados.

```
always @(posedge CLK) begin
 if (delay_done) begin
   LED_OUT <= 8'b0000_0000;
   if (direction == 1'b0) begin // derecha (0 -> 7)
     if (current_led_index == 7) begin // Si llegó al final
       direction <= 1'b1; // Cambiar dirección
       current_led_index <= 6; // retroceder desde LED6
     end else begin
       current_led_index <= current_led_index + 1; // Avanzar
     end
   end else begin // Moviendo a la izquierda (7 -> 0)
     if (current_led_index == 0) begin // Si llegó al final
       direction <= 1'b0; // Cambiar dirección a derecha
       current_led_index <= 1; // Empezar a avanzar desde LED1
     end else begin
       current_led_index <= current_led_index - 1; // Retroceder
     end
   end

   LED_OUT[current_led_index] <= 1'b1;
 end
end
```

Este bloque always, que se ejecutará en paralelo con el anterior (no después aunque en el código esté posterior, pues esto es hardware y los bloque se ejecutan en paralelo) hace la lógica de encender y apagar leds.

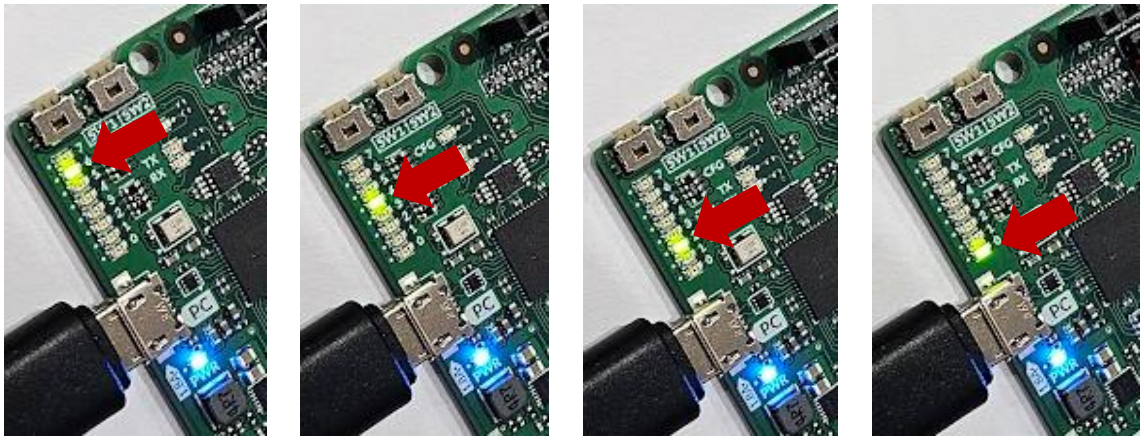
Básicamente, cuando detecta el delay_done, es decir, el ciclo de reloj donde se ha alcanzado el delay entre LEDS, realiza varias acciones. Primero pone LED_OUT todo a 0, lo que equivaldrá a apagar todos los leds (en vez de buscar apagar el que está encendido, esto asegura que todos se apagarán). Después el primer IF detecta si

las luces se encienden hacia la derecha (0-7) o a la izquierda (7-0). En caso de ir hacia la derecha, el segundo IF comprueba si ya estamos en el último LED (el 7), si estuviéramos ahí lo que hacemos es cambiar el sentido de la dirección e indicar que el LED a encender será el 7. El else siguiente comprueba lo mismo pero en sentid contrario.

```
    initial begin
        LED_OUT = 8'b0000_0001;
        current_led_index = 0;
        direction = 1'b0;
        delay_counter = 0;
        delay_done = 1'b0;
    end

endmodule
```

Finalmente hay un bloque initial que se ejecuta una sola vez, inicializando las variables.

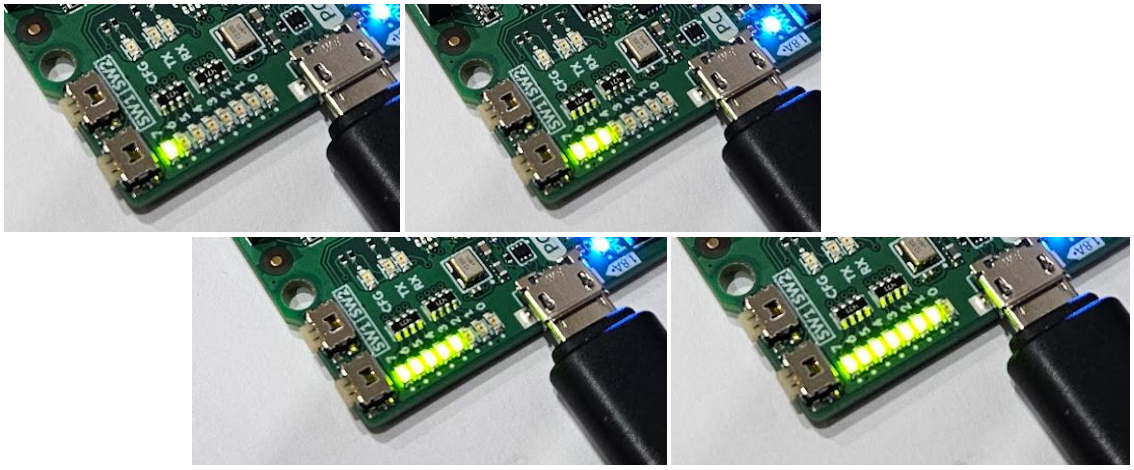
Si construimos el sintetizado `apio build`, y lo subimos a la placa `apio upload`, veremos el efecto de las deseadas luces.



Sobre el código que acabamos de explorar hay un aspecto curioso. La forma ne la que se apaga un led para encender el siguiente es utilizando la instrucción `LED_OUT <= 8'b0000_0000;` Parece un poco innecesario poner todos los bits a 0, cuando segúne l código solo hay 1 bit que pueda estar a 1, esto generará más hardware. ¿Por qué no usar `LED_OUT[current_led_index] <= 1'b0;` ?

Respuesta: porque no funcionaría como esperas.

El efecto que se generaría en las luces sería al de ir encendiéndolas todas, con un delay entre un LED y el siguiente, como muestra la siguiente imagen:



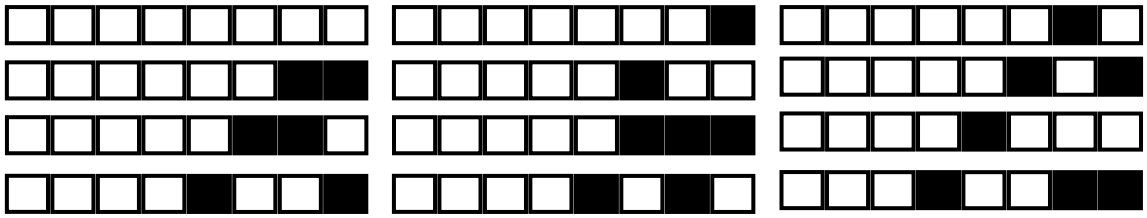
Recuerda que no estás ejecutando software, sino que este código generará un hardware que realiza acciones en cada ciclo de reloj, de echo en el código que genera el hardware, si te das cuenta se emplean asignaciones no bloqueantes, y afectarán a LED_OUT, direction y current_led_index de forma simultánea. Si sustituyes `LED_OUT <= 8'b0000_0000;` por `LED_OUT[current_led_index] <= 1'b0;` el código quedaría como se muestra a continuación:

```
...
LED_OUT[current_led_index] <= 1'b0;
  if (current_led_index == 7) begin // Si llegó al final
    ...
  end
end
LED_OUT[current_led_index] <= 1'b1;
```

Es decir, estarías asignando sobre el mismo bit el valor 0 y el valor 1. No olvides que las asignaciones se ejecutarán al final del flanco de reloj.

Ejemplo 8 – Contador binario con botón

En este ejemplo vamos a desarrollar un pequeño diseño hardware que haga de contador binario usando los leds. Es decir, los des deberán mostrar una cuenta de 0, 1, 2, 3... codificado en binario en los leds, es decir, deberemos ver algo parecido a esto (supón que estás viendo el array de leds):



Con la experiencia que ya tenemos, si pensamos en la lógica del circuito, seguro que por un lado debe actuar con los ciclos de reloj del de la placa, de forma que, si queremos que el contador, por ejemplo, cambie cada 1 seg., deberemos incluir algún tipo de contador que cuente hasta alcanzar el número de ciclos que ocupa un segundo (12 MHz en nuestr caso).

Por otro lado, necesitaremos también algún tipo de registro cuyos bits se identifiquen con cada LED, de forma que codificando los bits de dicho registro, se muestre el código de luces deseada. Para facilitar la cuenta, en un registro podemos almacenar también el valor del contador. Como tenemos 8 leds, ya podemos asegurar que la variable contador que almacenará le valor en decimal tendrá 8 bits también.

Finalmente, como queremos controlar que la pulsar un botón se reinicie la cuenta, deberemos incluir este control del estado del botón.

Nuestro código `binary_led_counter.v` queda de la siguiente manera:

```
module binary_led_counter (
    input wire CLK,
    input wire BTN,
    output reg [7:0] LED_OUT
);

    parameter CLK_FREQ = 12_000_000;
    parameter ONE_SEC_LIMIT = CLK_FREQ - 1;
    parameter SEC_COUNT_BITS = 24;
    reg [SEC_COUNT_BITS-1:0] sec_counter = 0;
    reg [7:0] binary_count = 0;

    initial begin
        LED_OUT = 8'b00000000;
        sec_counter = 0;
        binary_count = 0;
    end
```

```
    always @(posedge CLK) begin

        if (BTN == 1'b1) begin
            sec_counter <= 0;
            binary_count <= 0;
            LED_OUT <= 8'b00000000;
        end else begin
            if (sec_counter == ONE_SEC_LIMIT) begin
                sec_counter <= 0;
                binary_count <= binary_count + 1;
                LED_OUT <= binary_count + 1;
            end else begin
                sec_counter <= sec_counter + 1;
            end
        end
    end
endmodule
```

El código es relativamente sencillo. Nuestro hardware comprueba primero si está pulsado el botón, en tal caso, reinicia contadores y apaga los leds. En caso contrario, comprueba si el contador de tiempo ha llegado al límite (1 segundo representado por ONE_SEC_LIMIT). Si hemos alcanzado este límite, se reinicio el contador de segundos y tanto binary_count como LEDS se actualizan el siguiente valor. En caso contario, simplemente se incrementa el valor del contador de segundos.

El resultado es un gracioso contador binario.