WORKSHOP

Introducción al Diseño Digital con EDU-CIAA-FPGA

Aquí es donde todo comienza ...

Circuitos combinacionales

Contenido

- Introducción a Verilog.
  - Tipos de datos: wire, reg, constantes.
  - Operadores: bit-wise, reducción, lógicos, aritméticos, desplazamiento, relacionales, ternario, concatenación, replicación.
  - Asignaciones contínuas.
  - Asignaciones *Procedural*: if-else, for loop, while loop, case.
- Flujo de diseño.
- Ejemplo.
- Actividades propuestas.

Introducción a Verilog

Verilog

El lenguaje surge originalmente para SIMULAR circuitos, pero luego se comenzó a utilizar también para SINTETIZAR los mismos.

- De hecho, aplicado a las herramientas de desarrollo de FPGA, vamos a usar Verilog para las dos operaciones:
  - Síntesis del circuito: ‘archivo.v’ (Describiendo el circuito).
  - Simulación (test-bench): ‘archivo_tb.v’ (simula el comportamiento del circuito bajo prueba).
- Del conjunto del lenguaje HDL, solo un subconjunto muy pequeño “sintetiza” los circuitos.

Introducción a Verilog: tipos de datos

Tipos de datos

Nets: Representan las conexiones físicas entre componentes de hardware.

- Tipo Wire
- Otros que no usaremos (wand, supply0, ...)

- **Variables**: Representan almacenamiento abstracto en los módulos «de Comportamiento».
  - Tipo REG (Son sintetizables).
  - Tipo INTEGER (Los vemos mas adelante).
  - Tipo REAL, TIME, REALTIME (Solo en simulación).
- **Parameters**: *parameter, localparam.*

Tipos de datos: declaración

```
//Declaración de nets (wire)
wire my_wire;
wire [7:0] my_8bits_wire;
wire signed[7:0] my_signed_8bits_wire;
//Declaración de variables (reg)
reg my_reg;
reg [7:0] my_8bits_reg;
reg signed [7:0] my_signed_8bits_reg;
//Declaración de parámetro local:
localparam IDLE = 2'b00;
```

Tipos de datos

- Existen cuatro valores de datos ampliamente utilizados en la simulación y diseño de sistemas:
  - 0: 0 lógico o una condición falsa.
  - 1: 1 lógico o una condición verdadera.
  - Z: representa un estado de alta impedancia.
  - X: representa un valor indefinido.

Introducción a Verilog: operadores

Operadores

- Se utilizan tanto para asignar valores a nets (wire) como a variables (reg).
- Hay distintos tipos:
  - Bit-wise
  - Reduction
  - Logical
  - Arithmetic
  - Shift
  - Relational
  - Equality / inequality
  - Concatenation and replication
  - Conditional

Operadores: bit-wise

- Realizan la operación lógica indicada con todos los bits de los operandos (bit a bit).
- Ejemplos:
  - OR: 1100 | 1001 = 1101
  - AND: 1100 & 1001 = 1000
  - NOT: ~1100 = 0011
  - XOR: 1100 ^ 1001 = 0101
  - XNOR: 1100 ~^ 1001 = 1010

Operador	Función lógica
&	AND
|	OR
~	NOT
^	XOR
~^	XNOR

Operadores: reduction

- Realizan la operación lógica indicada con todos los bits del operando, dando como resultado un único bit (de ahí el nombre reducción).
- Ejemplos:
  - AND: &1100 = 0
  - NAND: ~&1100 = 1
  - OR: |0100 = 1
  - NOR: ~|0100 = 0
  - XOR: ^1011 = 1
  - XNOR: ~^1011 = 0

Operador	Tipo de reducción
&	AND
~&	NAND
|	OR
~|	NOR
^	XOR
~^	XNOR

Operadores: logical

- Similares a los operadores lógicos de C (if-else).
- Interpretan los operandos como True, False o X (unknown).
- Si un operando contiene solo ‘0’, se interpreta como False.
- Si un operando contiene al menos un ‘1’, se interpreta como True.
- Si el operando no tiene ‘1’s, pero tiene al menos un bit indeterminado (X), se interpreta como unknown.

Operador	Función lógica
!	NOT
&&	AND
||	OR

Operadores: arithmetic

- Realizan la operación aritmética entre los operandos. Hay que tener cuidado con el tamaño (en bits) del resultado para evitar overflow o underflow. **No todos son sintetizables.**
- Ejemplos:
  - 1001 + 0010 = 1011
  - 1001 + 1000 = **1**0001 (genera carry)
  - 110 – 001 = 101
  - 010 * 101 = 001010
  - 2**5 = 32
  - 10 / 3 = 3 (división entera)
  - 14 % 3 = 2

Operador	Tipo de operación
+	suma
-	resta
*	producto
**	potencia
/	división
%	módulo

Operadores: arithmetic

- Realizan la operación aritmética entre los operandos. Hay que tener cuidado con el tamaño (en bits) del resultado para evitar overflow o underflow. **No todos son sintetizables.**
- Ejemplos:
  - 1001 + 0010 = 1011
  - 1001 + 1000 = **1**0001 (genera carry)
  - 110 – 001 = 101
  - 010 * 101 = 001010
  - 2**5 = 32
  - 10 / 3 = 3 (división entera)
  - 14 % 3 = 2

Operador	Tipo de operación
+	suma
-	resta
*	producto
**	potencia
/	división
%	módulo

Operadores: shift

- Realizan desplazamiento de bits hacia derecha o izquierda manteniendo o no el signo del operando.
- Ejemplos:
  - 1001 << 2 = 0100
  - 1001 >> 2 = 0010
  - 1001 <<< 2 = 0100
  - 1001 >>> 2 = 1110 (si el resultado es signado)

Operador	Función lógica
<<	Shift left
>>	Shift rigth
<<<	Shift left artihmetic
>>>	Shift rigth arithmetic

Operadores: relational

- Realizan la comparación por mayor, menor, mayor o igual, menor o igual de los operandos, dando como resultado ‘0’ o ‘1’.
- Ejemplos:
  - 1001 < 0010 = 0
  - 1001 > 0010 = 1
  - 1001 <= 1001 = 1
  - 1001 >= 1100 = 0

Operador	Función lógica
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual

Operadores: equality / inequality

- Realizan la comparación por igual o distinto de los operandos, dando como resultado ‘0’ o ‘1’.
- Ejemplos:
  - (1001 == 0100) = 0
  - (1001 != 0100) = 1
  - (1001 == 01**x**0) = x
  - (1001 === 0100) = 0
  - (1001 !== 01**x**0) = 1
  - (01**x**0 === 01**x**0) = 1
  - (01**x**0 == 01**x**0) = x

Operador	Función lógica
==	Equality
!=	Inequality
===	Case equality
!==	Case inequality

Operadores: concatenation

- Realizan la concatenación de dos o más variables, nets o constantes.
- Ejemplos:
  - A = 1000
  - B = 11
  - C = 0000
  - {A,B,C} = 1000110000

Operador	Función lógica
{}	Concatenation

Operadores: Replication

- Da como resultado la replicación del operando por una cantidad definida.
- Ejemplos:
  - A = 101
  - {3{A}} = 101101101

Operador	Función lógica
{N{}}	Replication

Operadores: conditional

- Evalúa la condición que precede al signo de pregunta. Si es True, asigna el primer valor, de lo contrario asigna el segundo.
- Ejemplos:
  - mux_sel = 1
  - mux_out = (mux_sel) ? (in_0) : (in_1)

Operador	Función lógica
() ? () : ()	Conditional

Operadores: conditional

- Evalúa la condición que precede al signo de pregunta. Si es True, asigna el primer valor, de lo contrario asigna el segundo.
- Ejemplos:
  - mux_sel = 1
  - mux_out = (mux_sel) ? (in_0) : (in_1)

Aquí, mux_out toma el valor de in_0, ya que mux_sel = 1 (true)

Operador	Función lógica
() ? () : ()	Conditional

Introducción a Verilog: asignaciones contínuas

Asignaciones contínuas

- Representan una conexión permanente.
- Se utilizan con los tipos net (**wire**)

```
wire my_wire_1;
wire my_wire_2;
wire my_wire_3;
wire my_wire_4;

assign my_wire_1 = a & b;
assign my_wire_2 = ^c;
assign my_wire_3 = d + e;
assign my_wire_4 = f ? g : h;
```

Introducción a Verilog: procedurals

Asignaciones *procedural*

- Se utilizan comúnmente para asignar variables (reg).
- Admiten el uso de construcciones tipo *if-else, for loop, while loop, case,* entre otras, para describir el comportamiento de un circuito.
- Los bloques procedural utilizados frecuentemente son:
  - *always*
  - *initial*

Bloque *initial:*

- Utilizado principalmente en **simulaciones.**
- Se ejecuta por única vez al comienzo de la simulación.

```
reg reset_signal;

initial
begin
 reset_signal = 1'b1;
 #50.0; //wait for 50 time units
 reset_signal = 1'b0;
end
```

Bloque *always:*

- Utilizado tanto en simulación como en descripción de hardware.
- En descripción de hardware, se puede utilizar para modelar lógica combinacional y secuencial.
- Se ejecuta siempre que exista una transición en alguna de las señales presentes en su lista de sensibilidad:

```
//Combinational
//Full list
always@(a,b,e,f)
begin
 c = b | a;
 d = e + f;
end
```

```
//Combinational
//Full list with star (*)
always@(*)
begin
 c = b | a;
 d = e + f;
end
```

```
//Sequential at positive
//clock edge
always@(posedge clock)
begin
 q <= d;
end
endmodule
```

Procedurals: if-else

if-else:

```
always@(*)
begin
   if(a > b)
   begin
     greater_than = 1'b1;
     equal = 1'b0;
   end
   else if(a==b)
   begin
     greater_than = 1'b0;
     equal = 1'b1;
   end
   else
   begin
     greater_than = 1'b0;
     equal = 1'b0;
   end
end
```

Procedurals: *case-select*

case-select:

```
always@(*)
begin
 case(state)
   2'b00:
     c_out = (a > b);
   2'b01:
     c_out = (a == b);
   2'b10:
     c_out = (a < b);
   default:
     c_out = (a ^ b);
 endcase
end
```

Procedurals: loops

For loop:

```
reg [15:0] some_signal;
reg [4:0] ones_counter;
integer i;

always@(*)
begin
 ones_counter = 0;
 for(i=0; i<16; i=i+1)
 begin
   ones_counter = ones_counter + some_signal[i];
 end
end
```

while loop:

```
reg [15:0] some_signal;
reg [4:0] ones_counter;
integer i;

always@(*)
begin
 ones_counter = 0;
 i=0;
 while(i<16)
 begin
   ones_counter = ones_counter + some_signal[i];
   i=i+1;
 end
end
```

Introducción a Verilog:
module

Verilog module

Los módulos escritos en HDL **deben** verse como una colección de Hardware (conjunto de circuitos) y no como un algoritmo secuencial (Software)

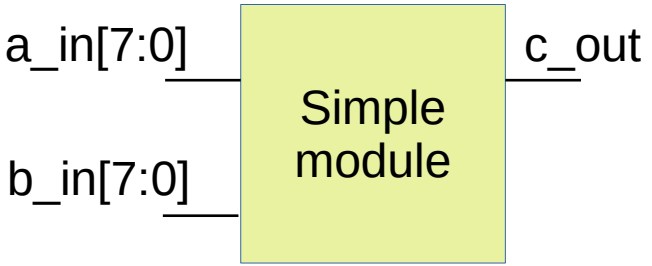
- Existen tres maneras de describir circuitos:
  - Por medio de **asignaciones continuas** a través de assign.
  - A través de **instanciación de módulos** (Vista estructural: Diagrama en bloques describiendo los componentes y sus interconexiones).
  - Construcción de **always block** (Vista de **comportamiento**: Describe la funcionalidad, trata al sistema como un objeto, pone el foco sobre la relación entre entradas y salidas)

Verilog module

```
module simple_module
#(
 parameter PORT_WIDTH     = 8
)
(
 input [PORT_WIDTH-1:0] a_in  ,
 input [PORT_WIDTH-1:0] b_in  ,
 output reg             c_out
);

always @(*)
begin
 c_out = a_in >= b_in;
end

endmodule
```



Verilog module

- Parámetos
  - Se utilizan en la instanciación de módulos para su configuración y no pueden ser modificados dentro del módulo una vez instanciado:

```
module cpu_uart
#(parameter CLKS_PER_BIT = 104)
(
    input wire clock_in ,
    input wire clr_in   ,
    output reg serial_out
);
```

```
cpu_uart cpu_dut
#(
    .CLKS_PER_BIT(150)
)
(
    .clock_in   (clock_in)   ,
    .clr_in     (clr_in)     ,
    .serial_out (serial_out)
);
```

Verilog module

- Parámetros locales
  - Se definen a través de la directiva « localparam »
  - Facilita la interpretación del código:

```
localparam NUM_BITS = 8    ;
localparam ADDR_LEN = 8    ;
localparam RAM_WORD = 16   ;
localparam IR_REG_LEN = 16 ;
```

Flujo de diseño

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.
- **Simulación y verificación del diseño** –> Se realizará con **iverilog** desde docker en VScode.

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.
- **Simulación y verificación del diseño** –> Se realizará con **iverilog** desde docker en VScode.
- **Síntesis**: Mapea a los recursos de la FPGA con herramientas libres o herramientas del fabricante. –> Se realizará con **yosys** desde docker en VScode.

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.
- **Simulación y verificación del diseño** –> Se realizará con **iverilog** desde docker en VScode.
- **Síntesis**: Mapea a los recursos de la FPGA con herramientas libres o herramientas del fabricante. –> Se realizará con **yosys** desde docker en VScode.
- **Place and Route**: posicionar y rutear los bloques programables dentro de la FPGA –> Se realizará con **nextpnr** desde docker en VScode.

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.
- **Simulación y verificación del diseño** –> Se realizará con **iverilog** desde docker en VScode.
- **Síntesis**: Mapea a los recursos de la FPGA con herramientas libres o herramientas del fabricante. –> Se realizará con **yosys** desde docker en VScode.
- **Place and Route**: posicionar y rutear los bloques programables dentro de la FPGA –> Se realizará con **nextpnr** desde docker en VScode.
- **Bitstream**: Generación del bit file para bajar a la FPGA –> Se realizará con **icepack** desde docker en VScode.

Flujo de diseño

- **Diseño en HDL** –> Se puede realizar con cualquier editor, nosotros vamos a usar **Visual Studio Code**.
- **Simulación y verificación del diseño** –> Se realizará con **iverilog** desde docker en VScode.
- **Síntesis**: Mapea a los recursos de la FPGA con herramientas libres o herramientas del fabricante. –> Se realizará con **yosys** desde docker en VScode.
- **Place and Route**: posicionar y rutear los bloques programables dentro de la FPGA –> Se realizará con **nextpnr** desde docker en VScode.
- **Bitstream**: Generación del bit file para bajar a la FPGA –> Se realizará con **icepack** desde docker en VScode.
- **Download to FPGA**: Bajar el bit file generado –> Se realizará con **iceprog** desde docker en VScode.

Ejemplo 1:
comparador de 1 bit

Ejemplo 1: comparador de 1 bit.

- Vamos a diseñar un comparador de 1 bit describiendo en detalle las partes del mismo, instrucciones utilizadas, etc.
- Para poder describir el hardware debemos conocer su funcionalidad es decir su tabla de verdad y su circuito.

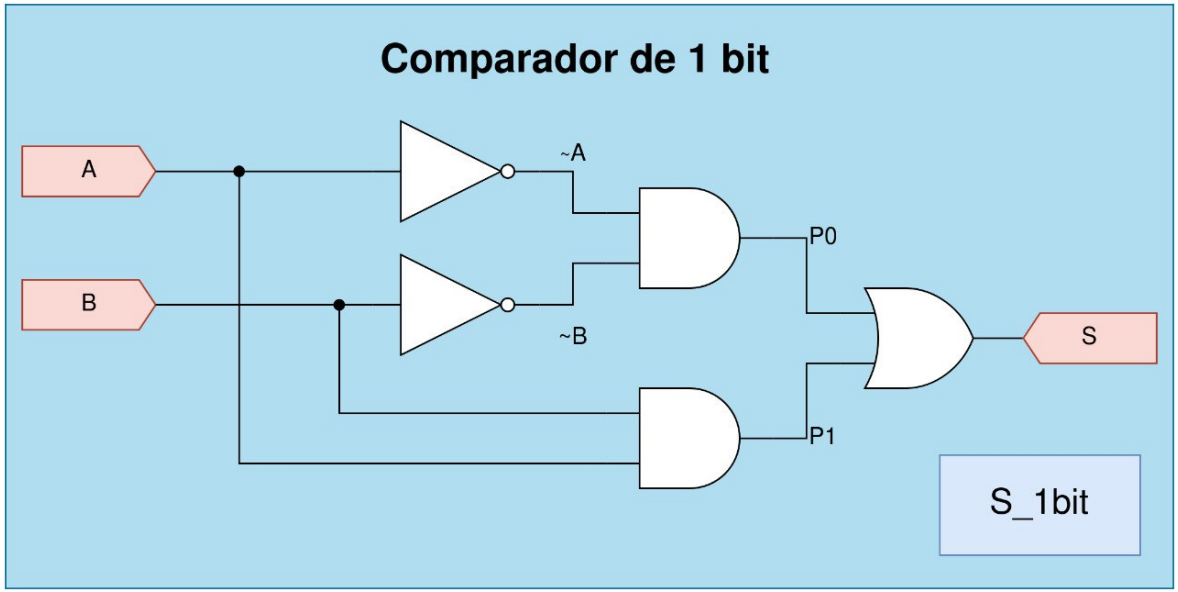
Ejemplo 1: comparador de 1 bit.

Vamos a comparar por igual y distinto, por mayor y menor se deja a los participantes.

A	B	A = B	A > B	A < B
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

Ejemplo 1: comparador de 1 bit.

Circuito del comparador de 1 bit que debemos describir.



Ejemplo 1: comparador de 1 bit.

```
module S_1bit
(
 input  wire A, B, //Declaración de los puertos de entrada
 output wire S     //Declaración del puerto de salida
);

 wire P0, P1;  //Declaración de nets internas (wires)

 assign S  = P0 | P1;  //Asignación continua S:salida
 assign P0 = A & B;
 assign P1 = ~A & ~B;

endmodule
```

Ejemplo 1: testbench

Ejemplo 1: testbench

- Los testbenches tienen como propósito verificar el funcionamiento de los bloques diseñados.
- Para tal fin, los testbenches excitan las entradas del módulo bajo prueba (DUT: *Device Under Test*) a través de estímulos generados por el propio testbench o leídos de archivos externos.

Ejemplo 1: testbench

En principio, el testbench puede verse como un “top level” sin puertos de entrada/salida, en el cual se **instancia** el DUT:

```
//Instanciación del Device Under Test (DUT)
S_1bit dut
(
  .A(A_in),   //Conexión de la entrada física A con la señal de estímulo
  .B(B_in),   //Conexión de la entrada física B con la señal de estímulo
  .S(AyB_out) //Conexión de la Salida del módulo
);
```

Ejemplo 1: testbench

Para excitar las entradas del DUT se definen variables tipo reg. Para leer sus salidas, se definen nets tipo wire:

```
// DUT Inputs
reg  A_in;      // Señal para estimular la entrada A
reg  B_in;      // Señal para estimular la entrada B
// DUT Outputs
wire AyB_out;  //Señal para capturar la salida del DUT
```

Ejemplo 1: testbench

Finalmente, para este ejemplo particular, se generan los estímulos dentro del testbench (bloque initial):

Ejemplo 1: testbench

```
initial
begin
  $display ("<< Comenzando el test >>");
  $dumpfile("test.vcd");   //Las formas de onda se guardarán en test.vcd
  $dumpvars(2, S_1bit_tb );//Se logean señales hasta 2 jerarquías por debajo del tb.
  #10;
  A_in = 1'b0;
  B_in = 1'b0;
  #100;   //Espera 100 unidades de tiempo.
  A_in = 1'b1;
  B_in = 1'b0;
  #100;
  A_in = 1'b0;
  B_in = 1'b1;
  #100;
  A_in = 1'b1;
  B_in = 1'b1;
  #100;
  $finish;
end
```

Ejemplo 1:
Simulación

Ejemplo 1: simulación

- Todos los procesos los vamos a realizar desde el VScode con la extensión de EDU-CIAA-FPGA.
- Es importante abrir Vscode en el directorio donde están los archivos fuentes del ejemplo.

Ejemplo 1: simulación

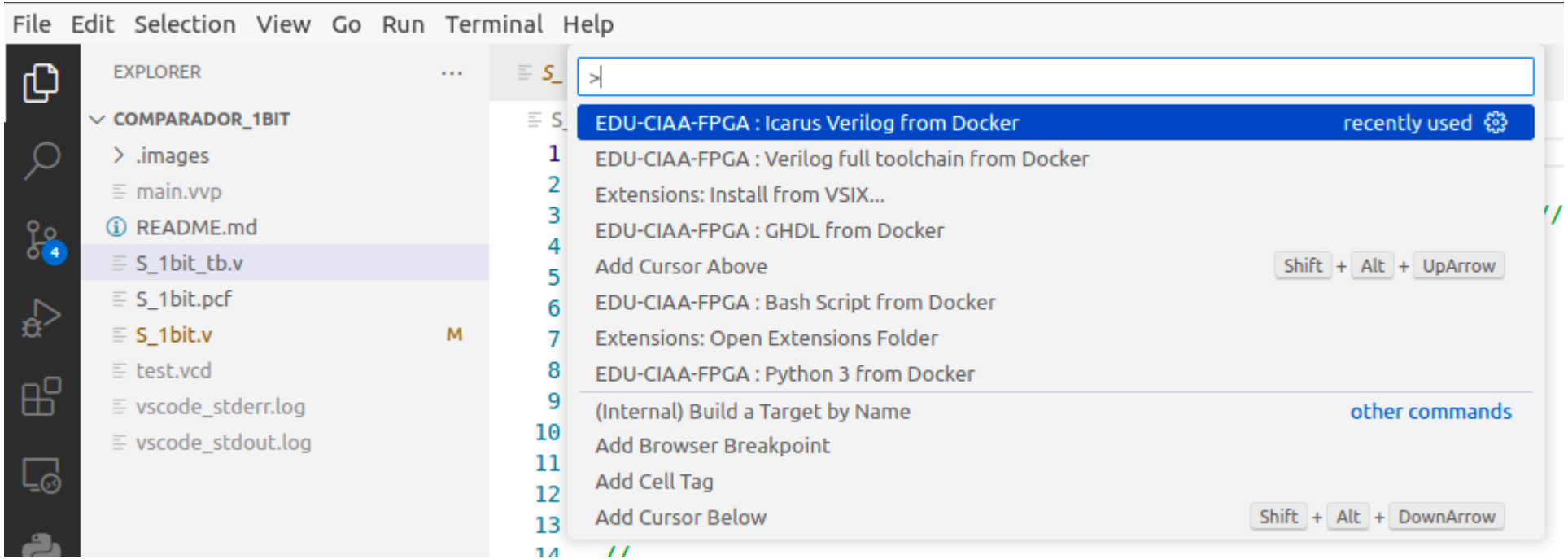
- Para abrir la paleta de comandos -> SHIFT + CTRL + P y buscar EDU-CIAA-FPGA.

- Las dos que vamos a utilizar son:
  - EDU-CIAA-FPGA: Icarus Verilog from docker
  - EDU-CIAA-FPGA: Verilog Toolchain from docker

Ejemplo 1: simulación

- EDU-CIAA-FPGA: Icarus Verilog from Docker (IVD)
  - Este comando verifica sintaxis, simula y visualiza formas de onda.
- EDU-CIAA-FPGA: Verilog Toolchain from (VTD) Docker
  - Este comando sintetiza en la FPGA el hardware descripto.

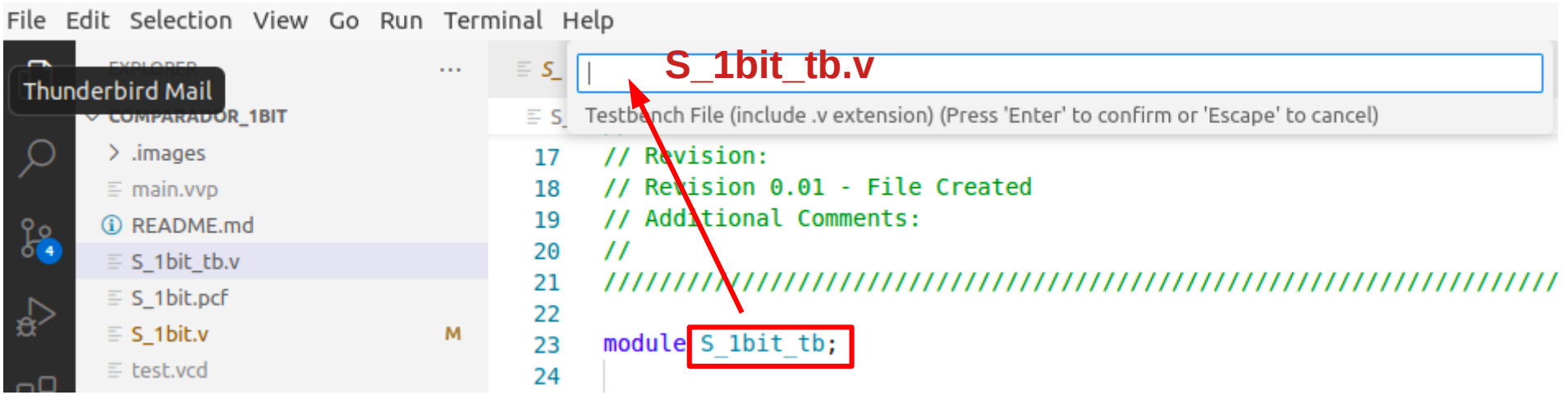
Ejemplo 1: simulación



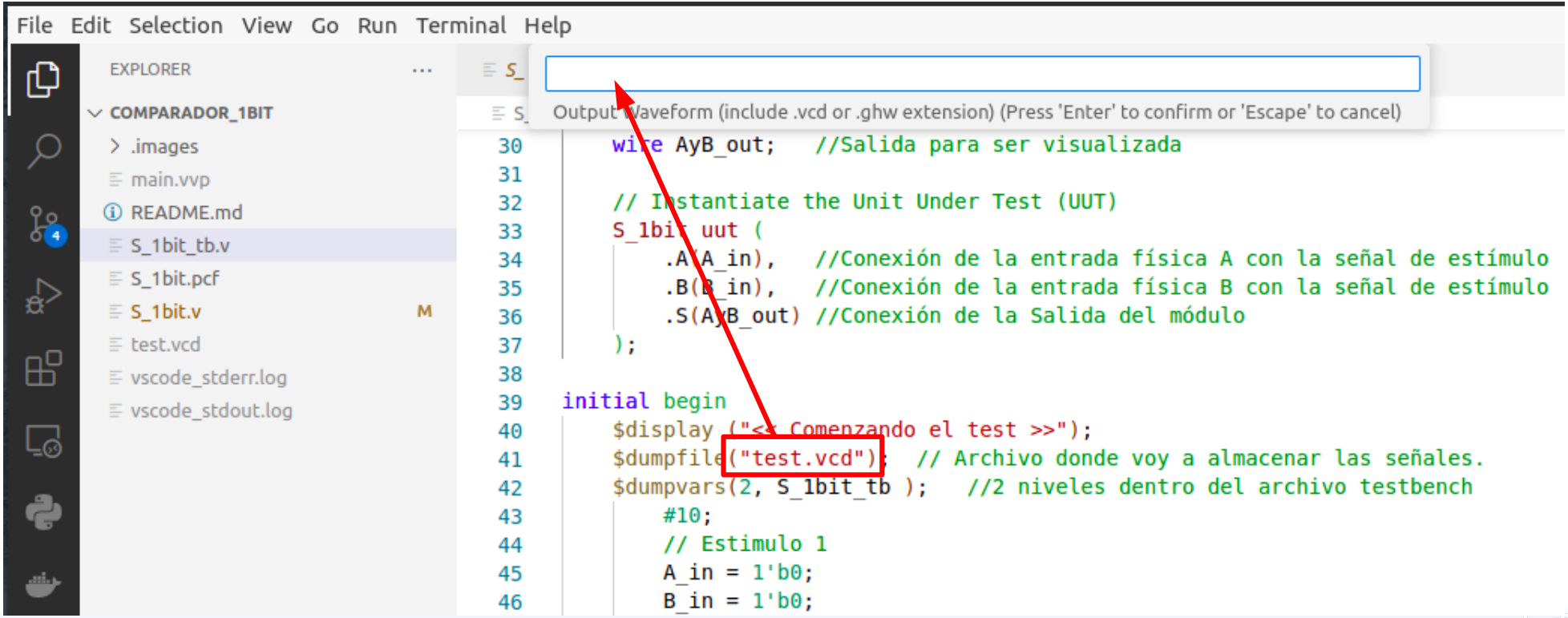
Ejemplo 1: simulación

Una vez que seleccionado el comando de simulación (IVD), es necesario indicar el archivo de formas de onda *(**test.vcd**)* y el nombre del testbench con la extensión (**S_1bit_tb.v**).

Ejemplo 1: simulación



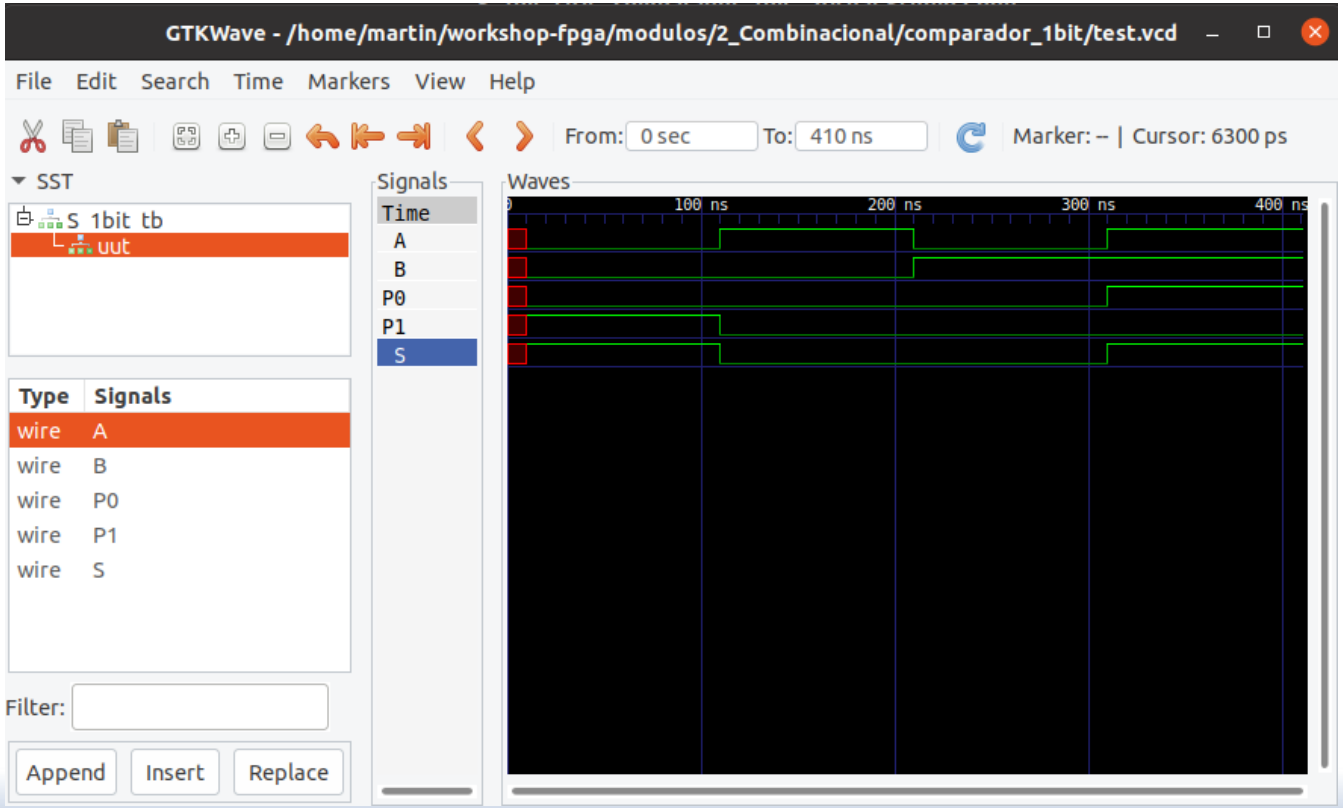
Ejemplo 1: simulación



Ejemplo 1: simulación

- Luego de ejecutar el comando IVD indicando los archivos necesarios, se compilará el test desde docker, ejecutará la simulación con la inyección de estímulos y guardará la evolución de las señales internas en el archivo *test.vcd*.
- Al finalizar el test, el comando IVD invoca al programa GTKWave para abrir el archivo *test.vcd* y visualizar las formas de onda.

Ejemplo 1: simulación



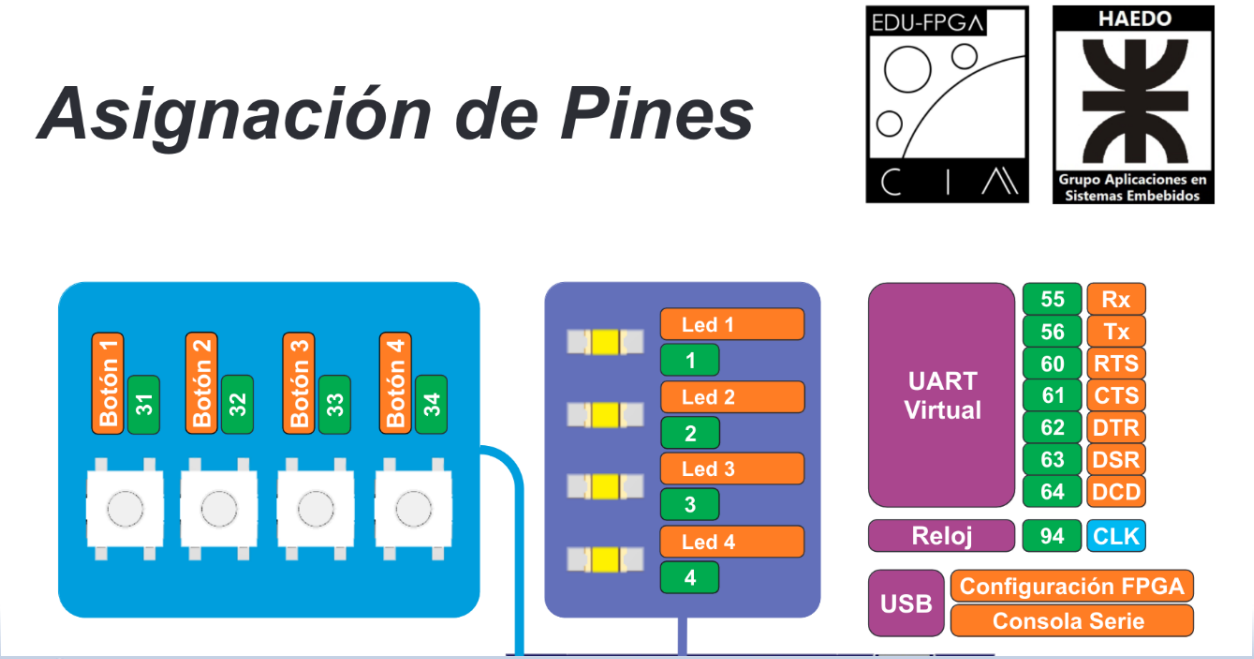
Ejemplo 1:
Síntesis e implementación

Ejemplo 1: síntesis e implementación.

- En este caso, dado que el testbench no implementa un auto-chequeo, la **verificación funcional** resulta de la **inspección visual** de las formas de onda.
- El paso siguiente, luego de verificar la funcionalidad, es **sintetizar** el diseño en hardware.
- Para esto último, será necesario indicar **cómo se rutean los puertos** de entrada / salida a los pines de la FPGA.

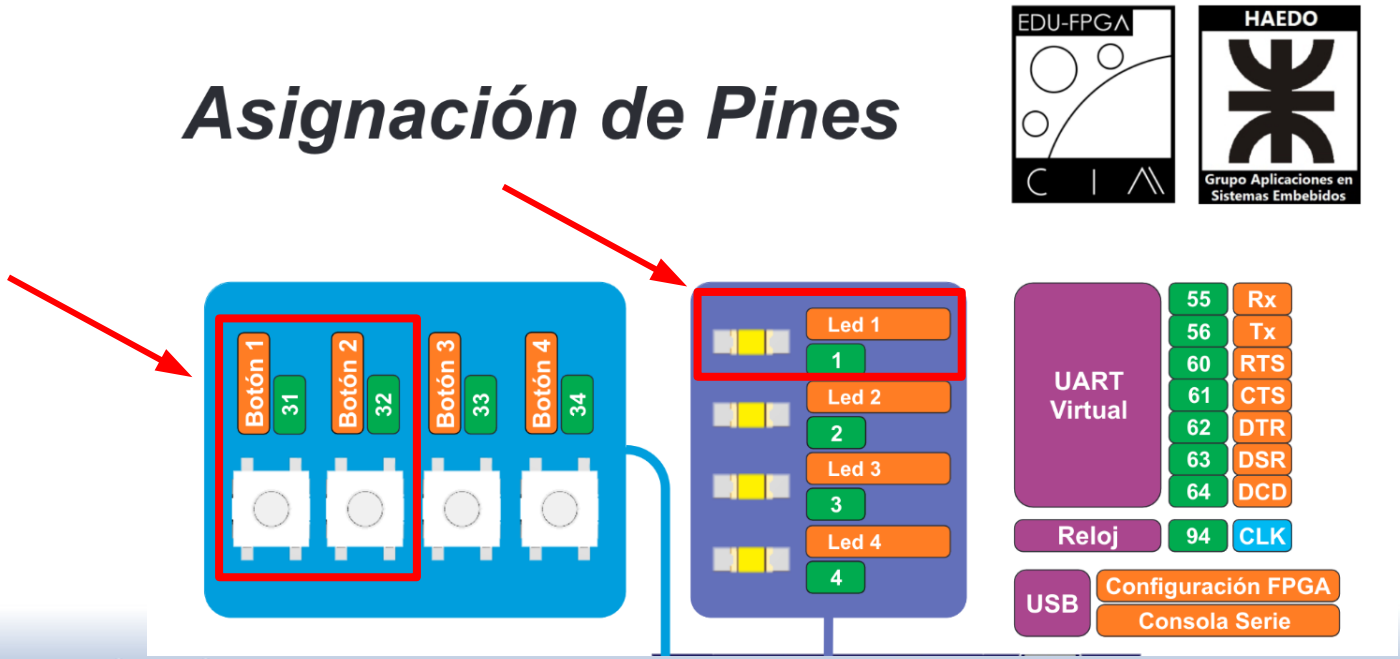
Ejemplo 1: síntesis e implementación.

Asignación de los pines necesarios



Ejemplo 1: síntesis e implementación.

Asignación de los pines necesarios



Ejemplo 1: síntesis e implementación.

- Como vimos en la imagen, podemos usar dos pulsadores como entradas del comparador y un led como salida para ver si compara correctamente.
- Esto se detalla en un archivo con extensión **.pcf** que indica al sintetizador cómo va a ser el conexionado del módulo.

Ejemplo 1: síntesis e implementación.

- Los archivos PCF o archivos de **constraint** deben contener las input / output del módulo y con qué pin de la FPGA se van a conectar.
- En este caso la entrada A va ir al pin 31 (pulsador 1).
- La entrada B al pin 32 (pulsador 2).
- La salida S al pin 1 (led 1).

Ejemplo 1: síntesis e implementación.

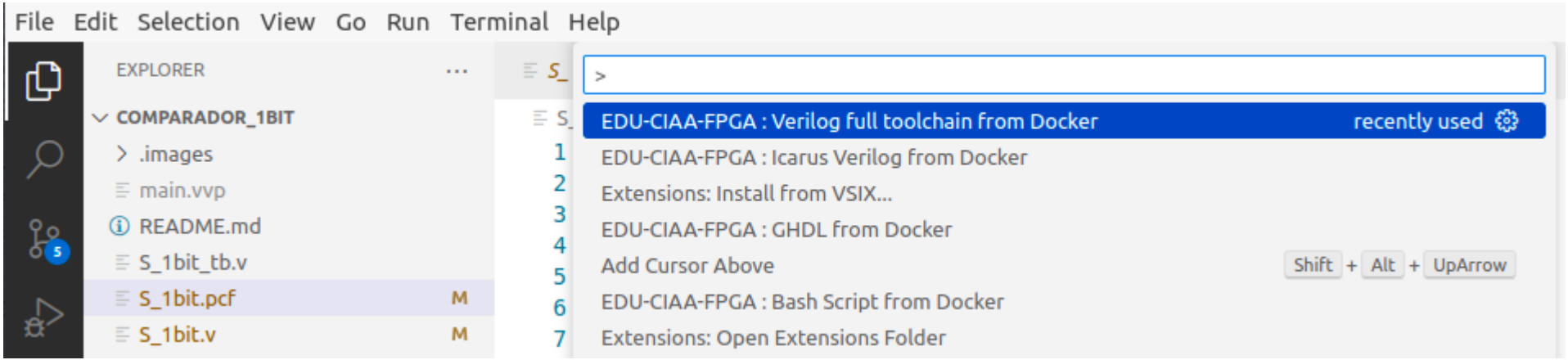
S_1bit.pcf

```
1   set_io A 31      #pin 31 Pulsador 1, se asigna a A
2   set_io B 32      #pin 32 Pulsador 2, se asigna a B
3   set_io S 1       #pin 1 led 1, se asigna a S
```

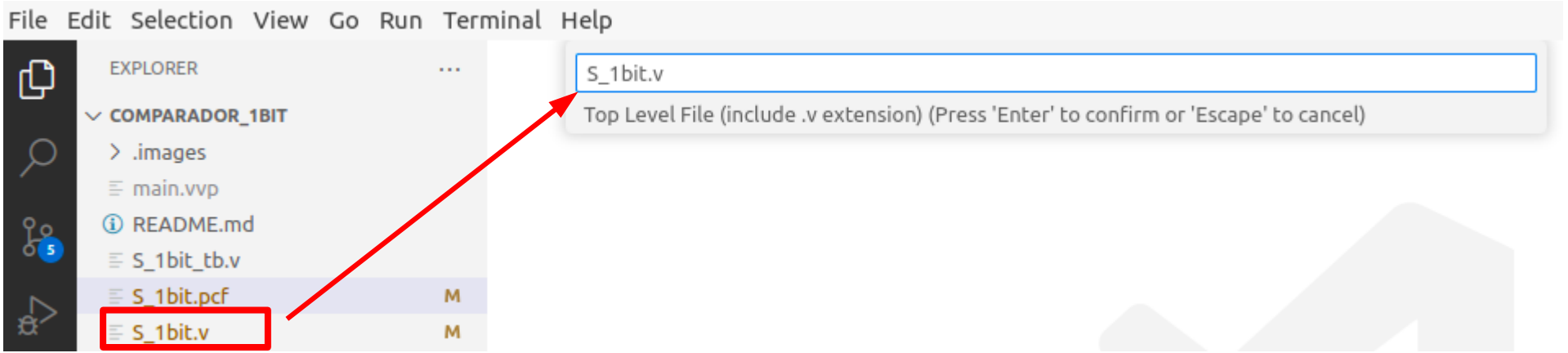
Ejemplo 1: síntesis e implementación.

- Una vez creado el archivo PCF (constraints), se puede continuar con la síntesis e implementación utilizando el comando:
  - EDU-CIAA-FPGA: Verilog Toolchain from docker (VTD)
- Para ejecutarlo, se abre la paleta de comandos y se busca el nombre del mismo.
- Este paso va a solicitar el módulo a implementar **(.v) y** los constraints **(.pcf)**.

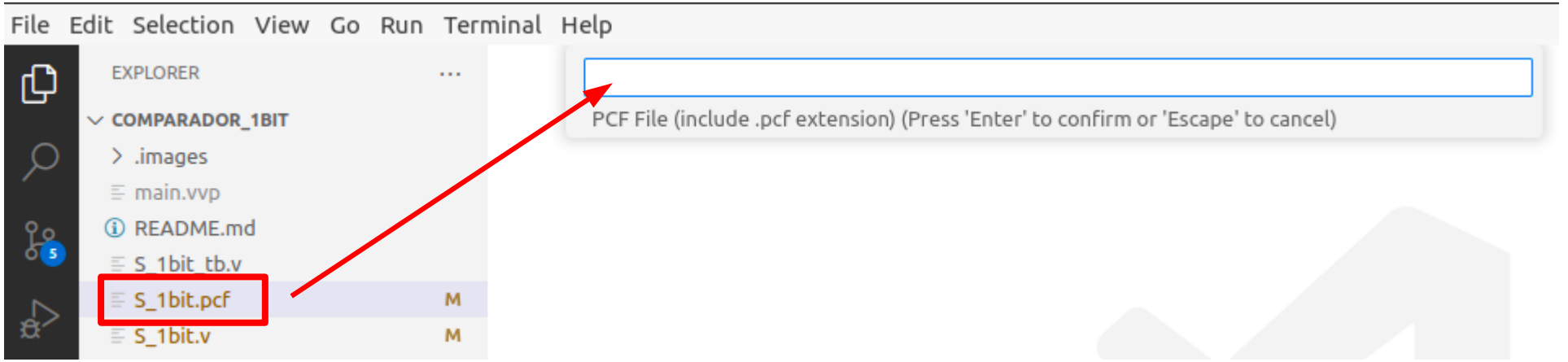
Ejemplo 1: síntesis e implementación.



Ejemplo 1: síntesis e implementación.



Ejemplo 1: síntesis e implementación.



Ejemplo 1: síntesis e implementación.

- Una vez sintetizado, VScode va a generar dos archivos de reporte donde se encuentran los distintos estados del diagrama de flujo del diseño y la síntesis del hardware.
- Estos archivos son: **vscode_stderr.log** y **vscode_stdout.log**.

Actividades propuestas

Ejercicio-1: MUX 2 a 1

- Sintetizar un multiplexor de 2 a 1 en la EDU-FPGA, utilizando como entrada 1 el pulsador 1, como entrada dos el pulsador 2, como selección el pulsador 3 y la salida el led 1

Ejercicio-2: Instruction decoder

Instruction code	Output
0000	load
0100	add
0001	bitand
0110	sub
1010	input
1110	output
1000	jump
1001	Jump conditional

¡Gracias por su atención!