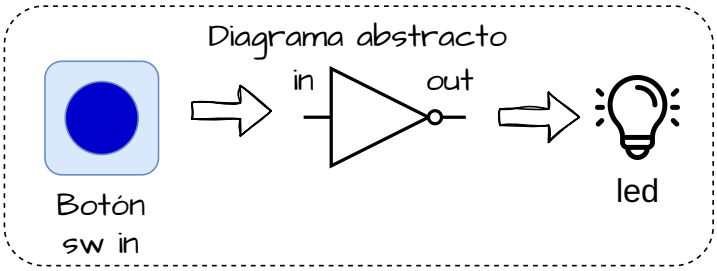
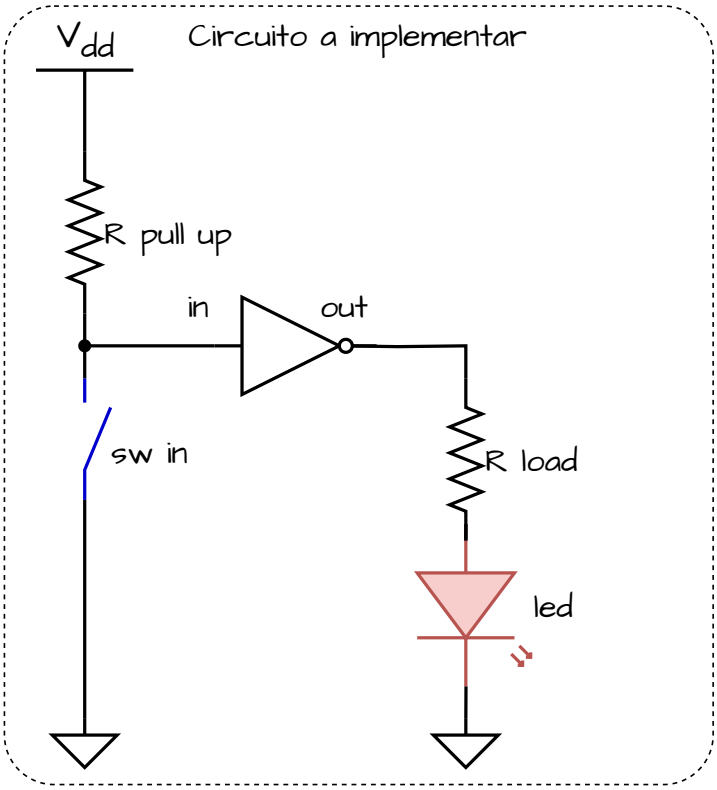
# Índice general

# Capítulo 1

# Aprende digital

## 1.1. Hola mundo: Compuerta implementada en una FPGA




| Funcionamiento esperado | | | |
|---|---|---|---|
| Botón sw in | In NOT | Out NOT | Led |
| Sin pulsar | 1 | 0 | Apagado |
| Pulsado | 0 | 1 | Encendido |

Figura 1.1: Circuito digital basado en la compuerta NOT

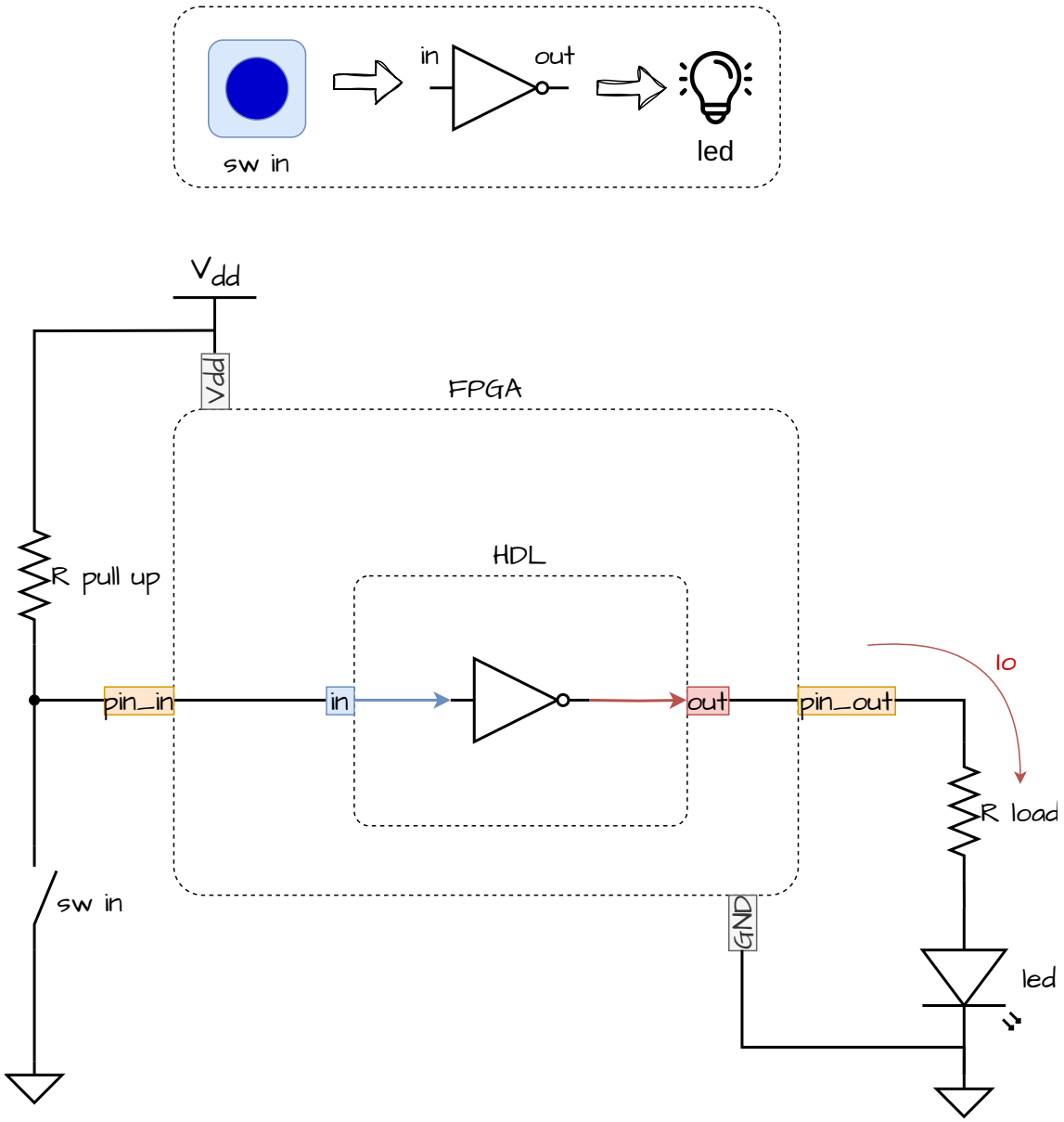



Figura 1.2: Circuito digital a implementar en la FPGA

### 1.1.1. Flujo de diseño del circuito hola mundo

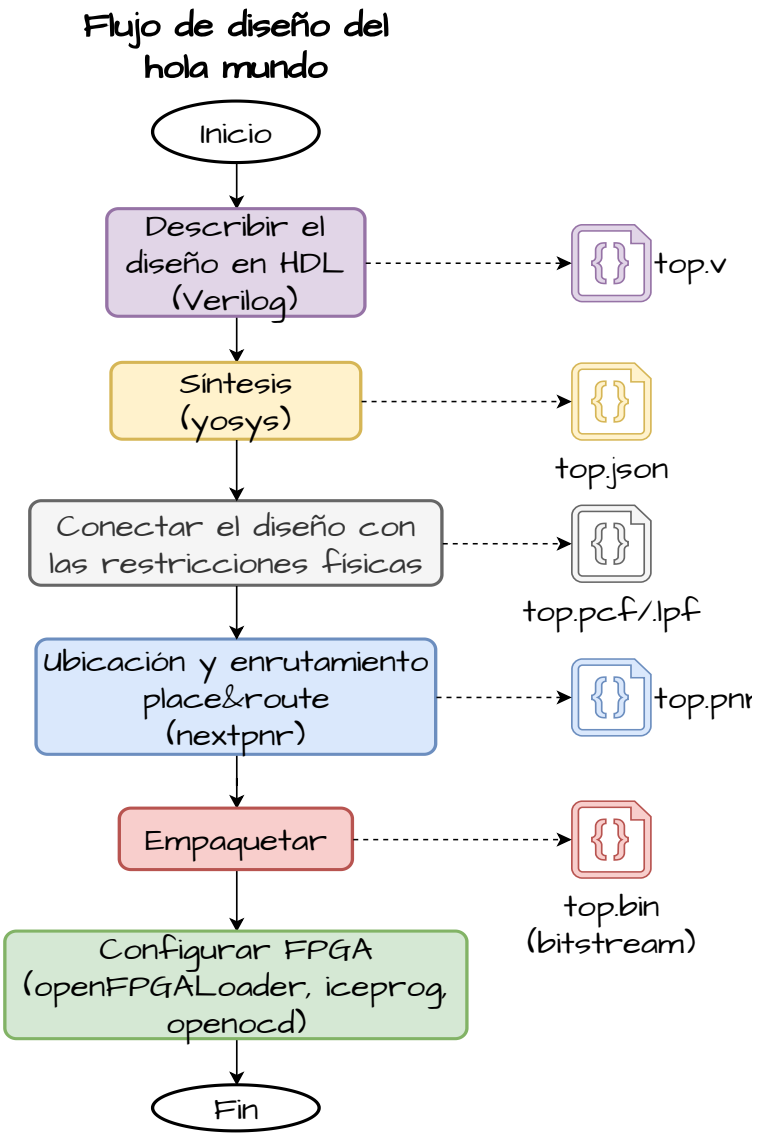



Figura 1.3: Flujo de diseño para la implementación del“hola mundo” en una FPGA con herramientas open-source

### 1.1.2. Descripción del diseño en el lenguaje HDL

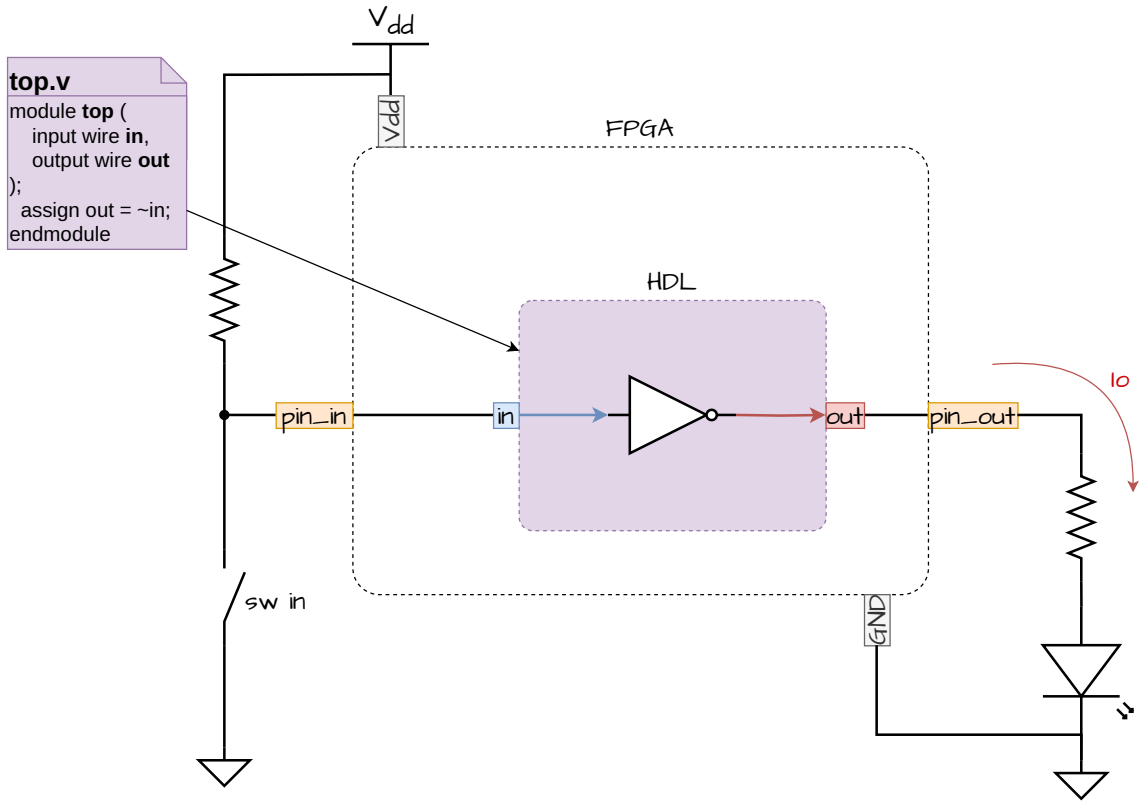



Figura 1.4: Descripción HDL del circuito a implementar en la FPGA

El lenguaje de HDL (Hardware Description Language) seleccionado para este ejemplo es Verilog.

Para iniciar el proceso de descripción, en su sistema GNU/Linux crear un directorio (carpeta) con el nombre representativo del diseño que está implementando, por ejemplo **holamundo** y dentro del directorio crear un archivo **top.v** con el contenido presentado en el código 1.1.

```
module top (
    // Definición del los puertos
    // Cable in asignado como entrada
    input  wire in,
    // Cable out assignado como salida
    output wire out
);
  // Descripción del comportamiento
  assign out = ~in;
endmodule
```

Código 1.1: Sistema digital descrito en Verilog sobre el operador NOT

### 1.1.3. Proceso de síntesis con Yosys

El proceso de síntesis dependerá de la tecnología de la FPGA con la que se cuente. Seleccione el comando correspondiente a la tecnología de la FPGA y ejecútelo en el *shell* de Linux al nivel donde está el contenido del directorio creado (**holamundo**) obtendrá como resultado un archivo **top.json** en el mismo directorio, puede ver el archivo ejecutando en la shell el comando **ls**.

Síntesis para FPGA ICE40

```
$ yosys -p "synth_ice40 -top top -json top.json" top.v
```

Síntesis para FPGA ECP5

```
$ yosys -p "synth_ecp5 -top top -json top.json" top.v
```

### 1.1.4. Diseño conectado a las restricciones físicas

Los archivos de configuración sobre las restricciones físicas permiten indicar de qué manera estará conectado el diseño en el contexto de la FPGA, por ejemplo, indicar por cual pin de la fpga se conecta una señal, indicaciones de tensiones umbrales, resistencias de pull-up entre otras. Dependiendo de la tecnología de la FPGA se podrá usar un archivo de configuración u otro (ver imagen 1.5).

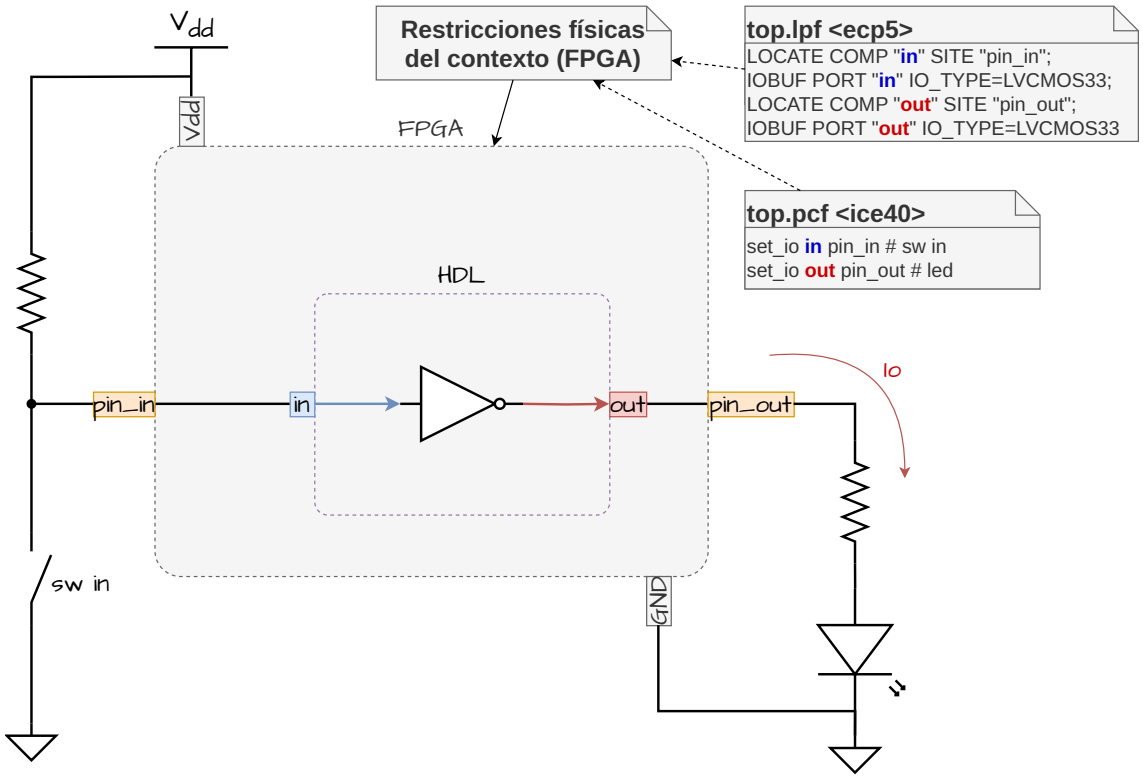



Figura 1.5: Diseño conectado a la FPGA a través de las restricciones físicas

Según la tecnología asociada a la fpga, en el código 1.2 o 1.3 tendrá que indicar el pin por donde se conecta la señal de su diseño. En el caso de código 1.3 deberá sustituir el nombre de **pin_in** y **pin_out** de las líneas 1 y 2 por el correspondiente pin de la FPGA (tecnología iCE40), mientras que en el código 1.2 deberá sustituir el nombre de **"pin_in"** y **"pin_out"** por el pin correspondiente de la FPGA (tecnología ECP5). En el directorio **holamundo** cree un archivo **top.pcf** (para ice40) o **top.lpf** (para ECP5) con el contenido que corresponda a la tecnología, use como referencia el código mostrado en 1.3 o 1.2.

```
# Lo que está seguido del signo # son comentarios.
# Filename: top.lpf
# Brief: Archivo de restricciones físicas para fpga de la familia ECP5.
```

```
# Pulsador en la entrada (in) del diseño
LOCATE COMP "in" SITE "pin_in"; # SUSTITUIR pin_in por el nombre del PIN de la FPGA a usar
IOBUF PORT "in" IO_TYPE=LVCMOS33;
# LED en la salida (out) del diseño
LOCATE COMP "out" SITE "pin_out"; # SUSTITUIR pin_out por el nombre del PIN de la FPGA a usar
IOBUF PORT "out" IO_TYPE=LVCMOS33
```

Código 1.2: Archivo top.lpf que representa las restricciones físicas para la FPGA ECP5

```
# Lo que está seguido del signo # son comentarios.
# Filename: top.pcf
# Brief: Archivo de restricciones físicas para fpga de la familia ICE40.

# Pulsador en la entrada (in) del diseño
set_io in pin_in # SUSTITUIR pin_in por el nombre/número del PIN de la FPGA
# LED en la salida (out) del diseño
set_io out pin_out # SUSTITUIR pin_out por el nombre/número del PIN de la FPGA
```

Código 1.3: Archivo top.pcf el cual representa las restricciones físicas para la FPGA ICE40

### 1.1.5. Ubicación y enrutamiento en FPGA

Dependiendo de la tecnología de la FPGA seleccione uno de los siguientes comandos y ejecútelo en el **shell** al nivel del contenido del directorio **holamundo**. Como resultado obtendrá un archivo **top.asc** o **top.pnr** el cual puede observar al ejecutar el comnado **ls** en el shell.

Place & route Colorlight ECP5

```
$ nextpnr-ecp5 --25k --package CABGA256 --speed 6 --json top.json --lpf top.lpf \
--freq 65 --textcfg top.pnr
```

Place & route para FPGA BlackICE iCE40-HX4K

```
$ nextpnr-ice40 --hx4k --package tq144 --json top.json --pcf top.pcf --asc top.pnr
```

Place & route para FPGA iCE40-HX8K Breakout Board

```
$ nextpnr-ice40 --hx8k --package ct256 --json top.json --pcf top.pcf --asc top.pnr
```

### 1.1.6. Empaquetar bitstream

Según la tecnología de la FPGA (ICE40 o ECP5) ejecute uno de los siguientes comandos en el *shell* de Linux al nivel del directorio **holamundo**, obtendrá como resultado un archivo **top.bin** (el bitstream de configuración de la FPPGA), el cual podrá observar al ejecutar el comando **ls**

Empaquetar bitstream para FPGA iCE40

```
$ icepack top.pnr top.bin
```

Empaquetar bitstream para FPGA ECP5

```
$ ecppack top.pnr top.bin
```

### 1.1.7. Configuración de FPGA

La configuración de la FPGA (enviar el bitstream a la FPGA) requiere de un puente de programación. Este puente puede ser implementado a través de un adaptador FTDI que haga uso del protocolo JTAG o SPI (dependiendo de la tecnología), también puede ser emulado por un microcontrolador u otro dispositivo, dependiendo del caso de la tarjeta de desarrollo que contiene la FPGA deberá verificar la conexión entre el puente y la FPGA y la conexión entre el puente y el *HOST* [1]. para tal finalidad puede visitar el apendice de configuración de tarjetas de desarrollo. Verificada la conexión, realice en la *shell* alguno de los siguientes comandos mostrados (según tecnología de la FPGA) al nivel del directorio **holamundo**. Este es el paso final del flujo de diseño y deberá verificar el funcionamiento del mismo interactuando con el pulsador y el LED.

Configurar FPGA BlackICE iCE40-HX4K

```
$ stty -F /dev/ttyACM0 raw
$ cat top.bin > /dev/ttyACM0
```

Configurar FPGA iCE40-HX8K Breakout Board

```
$ iceprog top.bin
```

Configurar FPGA Colorlight ECP5

```
$ openFPGALoader -c ft232RL top.bin
```

### 1.1.8. Configurando resistencias de pull-up desde FPGA

Las FPGA en sus archivos de restricciones físicas tiene entre sus diferentes opciones la posibilidad de conectar resistencias internas en modo pull-up y en algunos casos de pull-down (en las ECP5 es posible). Para el diseño implementado anteriormente, realice la modificación que se observa en la figura 1.6 en la cual se ha retirado la resistencia de pull-up externa y se reconfigura el archivo de restricciones físicas agregando entre los argumentos el valor para configurar las resistencias de pull-up internas. Realizada la configuración realice nuevamente el proceso de síntesis, place & route, empaquetado y configuración de la FPGA, finalmente pruebe el diseño comprobando que se comporta de la misma manera que con la resistencia externa.

[1] Se refiere al computador que tiene el sistema operativo que en este caso es Linux

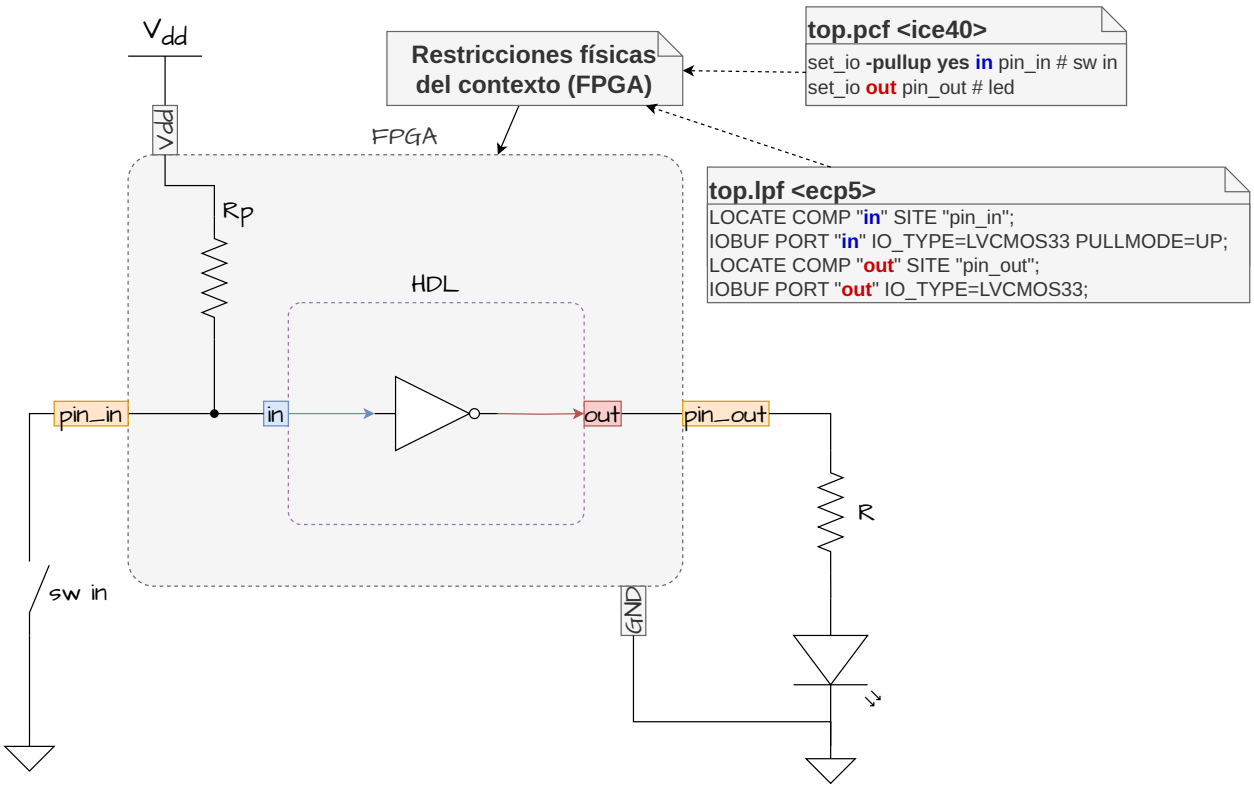



Figura 1.6: Diseño con resistencias de pull-up configuradas a través de restricciones físicas

### 1.1.9. Reto: Implementar otras compuertas en la FPGA

Para revisar los conceptos aprendidos en el proceso de diseño de circuitos digitales en FPGAs se propone que realice las modificaciones necesarias en los diferentes archivos para implementar otras compuertas como es el caso de la compuerta AND, entre otras, haga uso de referencia de la imagen 1.7, para realizar las pruebas del flujo de diseño.

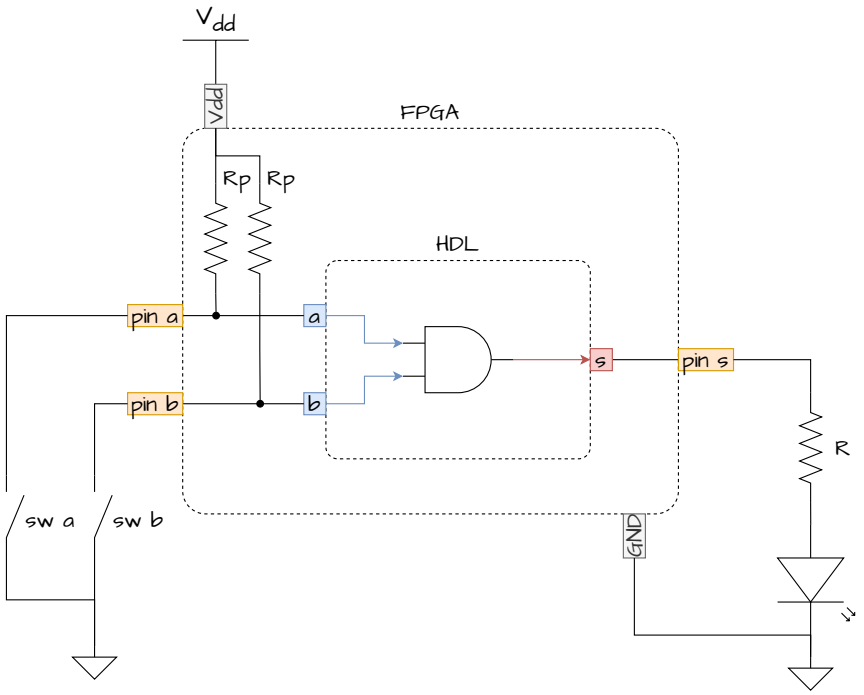



Figura 1.7: Circuito digital AND de ejemplo para implementar en FPGA

## 1.2. Simulación de un sumador de 4 bits y diagrama RTL

### 1.2.1. Flujo de simulación y creación de RTL

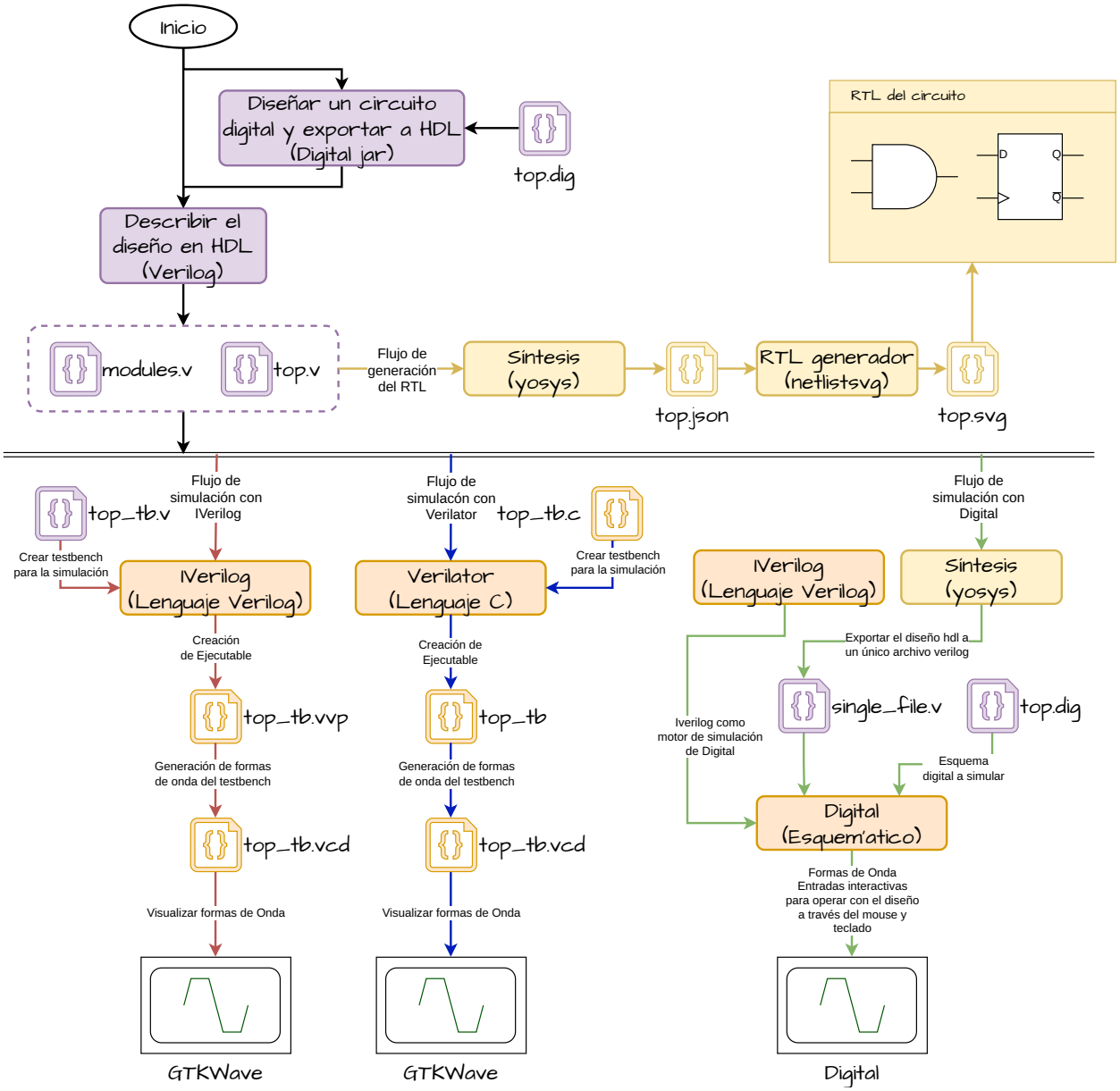



Figura 1.8: Flujo de simulación con herramientas opensource

### 1.2.2. Simulación del sumador medio en IVerilog

#### Descripción del sumador medio en Verilog

```
// filename: halfadder.v
// Declaración del módulo
module halfadder (
  input b, a,
```

```
  output co, s
);

// Declaración de señales
// ...

// Descripción del comportamiento
assign s = a ^ b; // s = a xor b
assign co = a & b; // a and b

endmodule
```

Código 1.4: Descripción Verilog de un medio sumador

### Generación del RTL del diseño a través de un Makefile

```
# filename: Makefile
# Identificador v para representar los archivos verilog
v=
# Identificador top para representar el top del RTL a visualizar
top=

rtl:
    # 1. Síntesis del diseño, si la sintaxis es correcta, se genera un archivo json que representa el diseño
    yosys -p 'read_verilog $v; prep -top $(top); hierarchy -check; proc; write_json $(top).json'
    # 2. Comando para generación del RTL en formato SVG (vectorial)
    netlistsvg $(top).json -o $(top).svg
    # 3. Visualizar el RTL con el visor de imagenes eog
    eog $(top).svg
```

Código 1.5: Comandos en archivo Makefile para la generación del RTL en formato SVG

Comado en terminal para crear el archivo SVG del RTL del sumador medio

```
$ make rtl v=halfadder.v top=halfadder
```

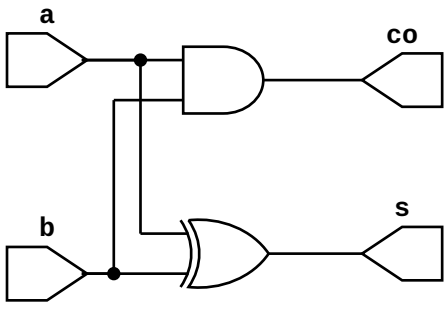



Figura 1.9: Representación en RTL del sumador medio

**Testbech y simulación en IVerilog del sumador medio**

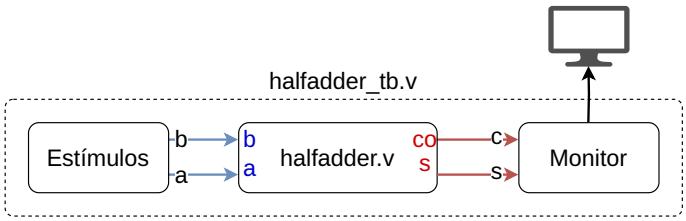



Figura 1.10: Representación del testbech requerido para el sumador medio

```
// filename: halfadder_tb.v
`include "./halfadder.v"

module halfadder_tb;

  // STIMULUS
  reg a = 0, b = 0;
  initial
  begin
     # 0 b = 0; a = 0;
     # 1 b = 0; a = 1;
     # 1 b = 1; a = 0;
     # 1 b = 1; a = 1;
     # 1 $finish(); // [stop(), $finish()]
  end

  // RESULT FOR DEVICE/DESIGN UNDER TEST
  wire c, s;

  // DEVICE/DESIGN UNDER TEST
  halfadder dut (.b(b), .a(a), .co(c), .s(s));

  // MONITOR
  initial
  begin
    $monitor("Time: %t, b = %d, a = %d => c = %d, s = %d",
      $time, b, a, c, s);
  end

endmodule
```

Código 1.6: Descripción del testbech para la simulación del sumador medio

```
# Identificador tb para el archivo verilog que contiene el testbench
tb=

sim:
	# 1. Crear el archivo .vvp ejecutable desde iverilog
	iverilog -o $(tb).vvp $(tb)
	# 2. Ejecuta el archivo .vvp para mostrar resultados
	vvp $(tb).vvp
```

Código 1.7: Comandos para agregar al Makefile que permiten la simulación con IVerilog

Ejecución de *make* para la simulación con IVerilog

```
$ make sim tb=halfadder_tb.v
```

**Al ejecutar el comando, se genera en la *Shell* el siguiente resultado:**
# 1. Crear el archivo .vvp ejecutable desde iverilog
iverilog -o halfadder_tb.v.vvp halfadder_tb.v
# 2. Ejecuta el archivo .vvp para mostrar resultados
vvp halfadder_tb.v.vvp
**Time: 0, b = 0, a = 0 =>c = 0, s = 0**
**Time: 1, b = 0, a = 1 =>c = 0, s = 1**
**Time: 2, b = 1, a = 0 =>c = 0, s = 1**
**Time: 3, b = 1, a = 1 =>c = 1, s = 0**
halfadder_tb.v:13: $finish called at 4 (1s)

### 1.2.3. Simulación del sumador completo en IVerilog con argumentos

```
// filename: fulladder.v
// Incluir otros archivos que hacen parte del diseño
`include "./halfadder.v"

module fulladder (
    // Inputs and output ports; 3 in, 2 out
    input  in_b,
    in_a,
    in_ci,
    output out_co,
    out_s
);

  // Declaración de señales
  wire s_s1_to_b_s2;  // Un cable del s del sumador 1 al a del sumador2
  wire co_s1_to_or;  // Un cable desde co del sumador 1 a la compuerta or
  wire co_s2_to_or;  // Un cable desde co del sumador 2 a la compuerta or

  // Declaración de submodules
  // halfadder(b,a,co,s)
  halfadder halfadder1 (
      in_b,
      in_a,
      co_s1_to_or,
      s_s1_to_b_s2
  );
  halfadder halfadder2 (
      s_s1_to_b_s2,
      in_ci,
      co_s2_to_or,
      out_s
  );

  // Descripción del comportamiento
  assign out_co = co_s1_to_or | co_s2_to_or;  // co_s1 or co_s2

endmodule
```

Código 1.8: Descripción HDL de un sumador completo

Comado en terminal para crear el archivo SVG del RTL del sumador completo

```
$ make rtl v=fulladder.v top=fulladder
```

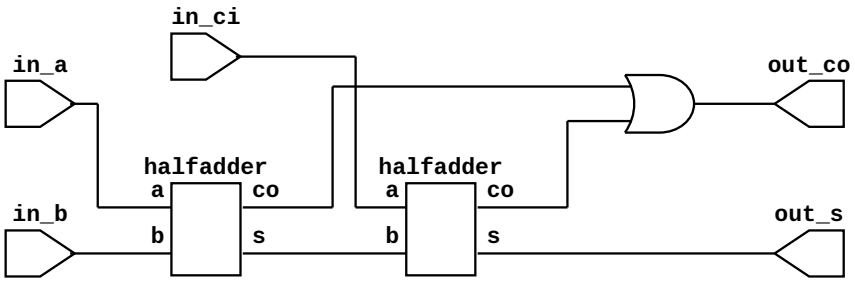



Figura 1.11: Sumador completo representado en RTL

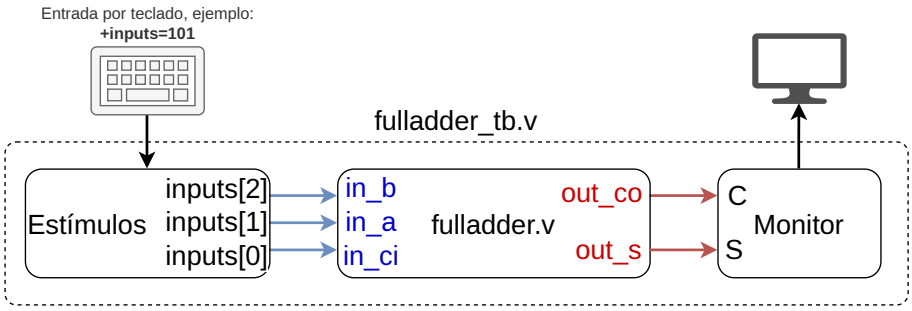



Figura 1.12: Representación del testbech requerido para el sumador completo

```
// filename: fulladder_tb.v
`include "./fulladder.v"

module fulladder_tb;

  // STIMULUS ARGS
  reg [2:0] inputs;
  initial
  begin
    if(! $value$plusargs("inputs=%b", inputs))
    begin
      $display("Please specify +inputs=<value> to start."); $finish;
    end
    else
    begin
      $display("Entradas:\tin_b=%b, in_a=%b, in_ci=%b", inputs[2], inputs[1], inputs[0]);
    end
  end
```

```
  // RESULT FOR DEVICE/DESIGN UNDER TEST
  wire c, s;

  // DEVICE/DESIGN UNDER TEST
  fulladder dut (.in_b(inputs[2]), .in_a(inputs[1]), .in_ci(inputs[0]),
    .out_co(c), .out_s(s));

  // MONITOR
  initial
  begin
    wait (c | s) $display("Salidas:\tout_co=%b, out_s=%b", c, s);
    #1 $finish;
  end

endmodule
```

Código 1.9: Testbech para la simulación del sumador completo con uso de argumentos

Comandos en el *Shell* para ejecutar la simulación del diseño con argumentos.

```
$ make sim tb=fulladder_tb.v
```

Si el comando se ejecutó correctamente dará como resultado la generación de un archivo **fulladder_tb.v.vvp** el cual podrá observar al ejecutar el comando `$ ls`

El archivo de extensión **.vvp** es un ejecutable que se puede lanzar cuantas veces sea requerido. A continuación un ejemplo, estimulando el diseño con un valor en **+inputs**:

```
$ vvp fulladder_tb.v.vvp +inputs=111
```

**Entradas: in_b=1, in_a=1, in_ci=1**
**Salidas: out_co=1, out_s=1**
fulladder_tb.v:34: $finish called at 1 (1s)

### 1.2.4. Simulación de sumador completo de 4bits en IVerilog y GTKWave

```
// filename: fulladder_4bits.v
`include "./fulladder.v"

module fulladder_4bits (
// Inputs and output ports
  input wire b3, b2, b1, b0,
    a3, a2, a1, a0,
    ci,
  output wire co,
  s3, s2, s1, s0
);

// Declaración de señales [reg, wire]
wire co0, co1, co2;

// Descripción del comportamiento
fulladder fulladders0(
  .in_b(b0), .in_a(a0), .in_ci(ci),
  .out_co(co0), .out_s(s0)
```

```
);

fulladder fulladders1(
  .in_b(b1), .in_a(a1), .in_ci(co0),
  .out_co(co1), .out_s(s1)
);

fulladder fulladders2(
  .in_b(b2), .in_a(a2), .in_ci(co1),
  .out_co(co2), .out_s(s2)
);

fulladder fulladders3(
  .in_b(b3), .in_a(a3), .in_ci(co2),
  .out_co(co), .out_s(s3)
);

endmodule
```

Código 1.10: Descripción en Verilog para el sumador de 4 bits

Comado en terminal para crear el archivo SVG del RTL del sumador completo de 4 bits

```
$ make rtl v=fulladder_4bits.v top=fulladder_4bits
```

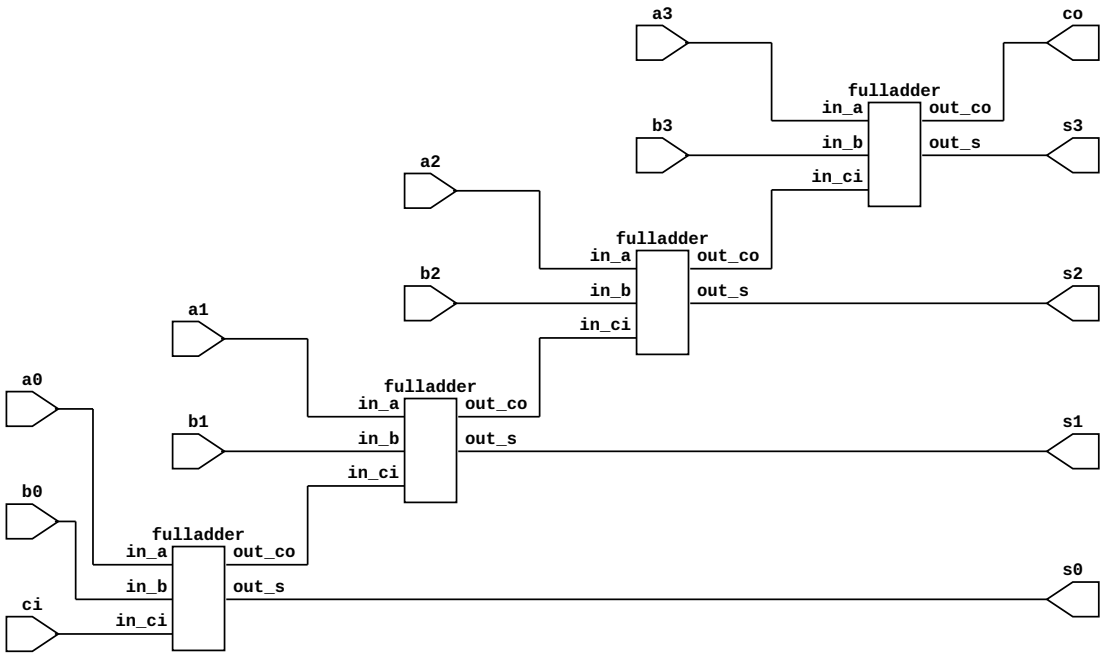



Figura 1.13: Representación del RTL del sumador completo de 4 bits

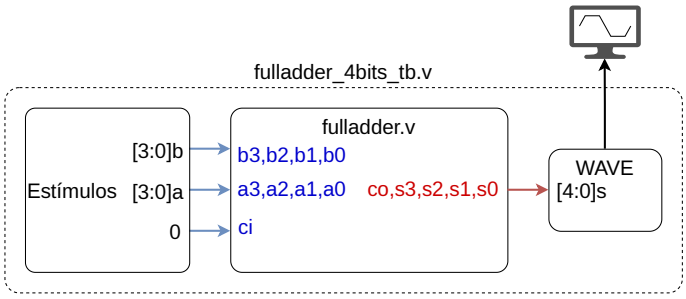



Figura 1.14: Representación del testbech requerido para el sumador completo de 4 bits

```
// filename: fulladder_4bits_tb.v
`include "./fulladder_4bits.v"

module fulladder_4bits_tb;
  reg [3:0] a;
  reg [3:0] b;

  integer i;
  integer j;

  initial
  begin
    a = 0;
    b = 0;
    for (i=0; i<16; i=i+1)
    begin
      for (j=0; j<16; j=j+1)
      begin
        a = j;
        b = i;
        #1;
      end
    end
    $finish();
  end

  // RESULT FOR DEVICE/DESIGN UNDER TEST
  wire [4:0] s;

  // DEVICE/DESIGN UNDER TEST
  fulladder_4bits dut (
    .b3(b[3]), .b2(b[2]), .b1(b[1]), .b0(b[0]),
    .a3(a[3]), .a2(a[2]), .a1(a[1]), .a0(a[0]), .ci(1'b0),
    .co(s[4]), .s3(s[3]), .s2(s[2]), .s1(s[1]), .s0(s[0]));

  // WAVES IN VCD TO OPEN IN GTKWAVE
  initial
  begin
    $dumpfile("fulladder_4bits_tb.vcd");
    $dumpvars(0, testbech);
  end
endmodule
```

Código 1.11: Testbech para la sumulación del sumador de 4 bits con IVerilog

Comando *make* ejecutado en el *Shell* para la simulación del diseño

```
$ make sim tb=fulladder_4bits_tb.v
```

Comando en el *Shell* para ver las formas de onda en *GTKWave*

```
$ gtkwave fulladder_4bits_tb.vcd
```

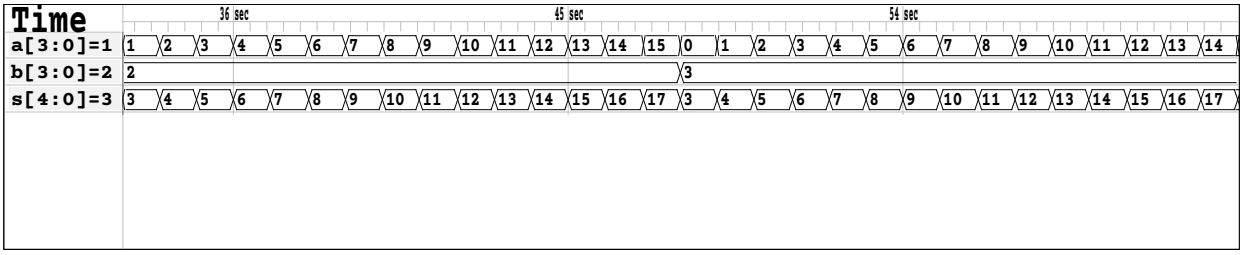



Figura 1.15: Formas de onda resultado de la simulación y visualización en GTKWave

### 1.2.5. Simulación de sumador completo de 4bits en *Digital* con IVerilog

```
# El siguiente comando permite convertir el diseño que se encuentra en distintos archivos
# Verilog a un solo archivo compacto de Verilog indicando el módulo TOP
design2VerilogFile:
	yosys -p 'read_verilog $v; hierarchy -top $(top); write_verilog $(top)_top.v'
```

Código 1.12: Comandos para agregar al final del archivo Makefile para convertir el diseño en un solo archivo Verilog

Comando *make* para ejecutar en el *Shell* y convertir el diseño a un solo archivo Verilog

```
$ make design2VerilogFile v=fulladder_4bits.v top=fulladder_4bits
```

```
$ ls
```

halfadder.v fulladder.v fulladder_4bits.v **fulladder_4bits_top.v** Makefile

Comandos para realizar la simulación en *Digital* con *IVerilog* para un proyecto descrito en Verilog

Abra una **nueva shell** (terminal) y en ella ubique el directorio que contiene el proyecto a simular, por ejemplo, suponiendo que el proyecto se encuentra en el directorio **/home/<user>/proyectos/fulladder_4bits**, se puede ir al directorio y listar el contenido como sigue:

```
$ cd ~/proyectos/fulladder_4bits/
```

```
$ ls # A continuación se lista los archivos que contiene ese directorio
```

halfadder.v fulladder.v fulladder_4bits.v fulladder_4bits_top.v Makefile

Ahora, desde el directorio del proyecto, ejecute los siguientes dos comandos:

```
$ conda activate digital # Iniciar variable de entorno para usar iverilog en Digital.sh
```

```
$ Digital.sh # Iniciar el simulador Digital
```

En *Digital* podrá realizar diferentes simulaciones de sistemas digitales y podrá integrar *IVerilog* para agregar sistemas descritos en Verilog a la simulación. Lo que primero se debe realizar es la revisión de la documentación oficial de Digital [2] la cual se encuentra en distintos lenguajes. Para la simulación del sumador completo de 4 bits, deberá crear un nuevo proyecto en *Digital* y guardarlo en el mismo directorio donde se encuentra el proyecto a simular, el cual contiene los diferentes archivos Verilog. Verifique en el directorio que se haya creado en él un archivo con extensión **.dig**. A continuación, en la barra de herramientas de *Digital*, ubique la herramienta **Components ->Misc ->VHDL/Verilog ->External File** y agregue el componente en el workspace (área de trabajo) de *Digital*, seguido, de clic derecho en el símbolo que representa al componente y edite los campos como se aprecia en la figura 1.16, como también se explica a continuación:

- **Label: fulladder_4bits**. Este representa el nombre del **module top** a simular que se encuentra en el archivo de verilog **fulladder_4bits_top.v**
- **Width: 4**. Se refiere al tamaño de la caja (solo es un tema visual en *Digital*) que representa al sistema a simular.
- **Inputs: b3,b2,b1,b0,a3,a2,a1,a0,ci**, Representa los puertos de entrada del módulo **fulladder_4bits** y debe coincidir con los indicados en el archivo Verilog.
- **Outputs: co,s3,s2,s1,s0**, Representa los puertos de salida del módulo, los cuales deben coincidir con los indicados en el archivo verilog.
- **Program code: ./fulladder_4bits_top.v**, Nombre del archivo Verilog con ruta relativa (./) del archivo a simular.
- **Application: IVerilog**, Se indica el simulador IVerilog a integrar en la simulación de *Digital*.
- **Check**, Se debe ejecutar el Checking (oprimir el botón *Check*) después de realizar la configuración manual indicada, si algún campo no coincide con los indicados en el módulo descrito en el top del archivo Verilog se generará una ventana emergente con el error generado. Lo ideal es que al hacer el Checking, no genere ninguna ventana emergente, lo cual indicará que se ha realizado el Checking sin errores.

[2] Visite el sitio `https://github.com/hneemann/Digital/releases` y descargue el documento PDF en el lenguaje que prefiera para que comprenda cómo funciona Digital

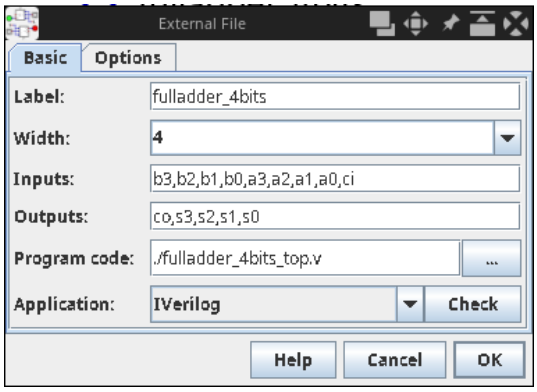



Figura 1.16: Configuración manual requerida para realizar la simulación en *Digital* del sumador de 4 bits

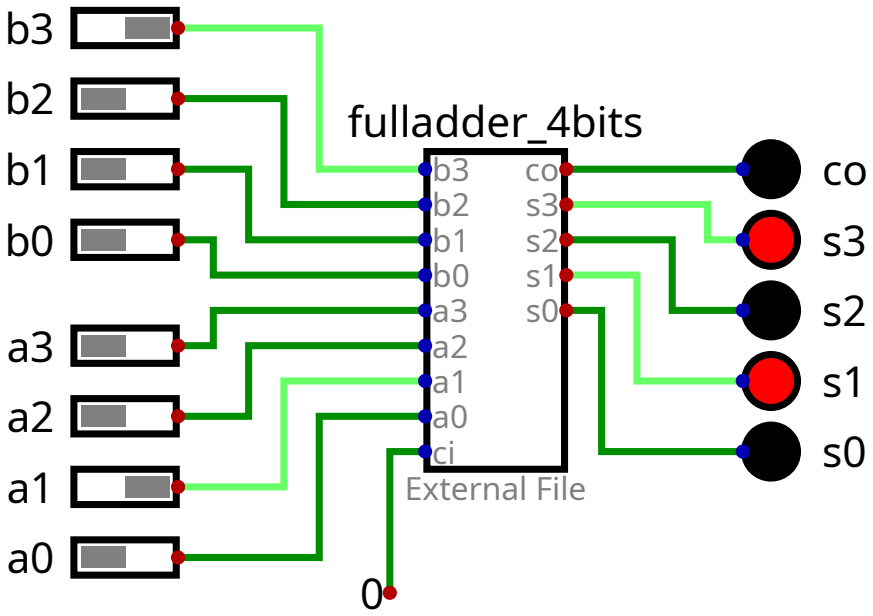



Figura 1.17: Diagrama digital creado en *Digital* para la simulación del sumador de 4 bits

```
# filename: Makefile
# Identificador v para representar los archivos verilog
v=
# Identificador top para representar el top del RTL a visualizar
top=
# Identificador tb para el archivo verilog que contiene el testbench
tb=

rtl:
    # 1. Síntesis del diseño, si la sintaxis es correcta, se genera un archivo json que representa el diseño
    yosys -p 'read_verilog $v; prep -top $(top); hierarchy -check; proc; write_json $(top).json'
    # 2. Comando para generación del RTL en formato SVG (vectorial)
    netlistsvg $(top).json -o $(top).svg
    # 3. Visualizar el RTL con el visor de imagenes eog
    eog $(top).svg

sim:
```

```
____# 1. Crear el archivo .vvp ejecutable desde iverilog
____iverilog -o $(tb).vvp $(tb)
____# 2. Ejecuta el archivo .vvp para mostrar resultados
____vvp $(tb).vvp

# El siguiente comando permite convertir el diseño que se encuentra en distintos archivos
# Verilog a un solo archivo compacto de Verilog indicando el módulo TOP
design2VerilogFile:
____yosys -p 'read_verilog $v; hierarchy -top $(top); write_verilog $(top)_top.v'

clean:
____# Borrar los objetos creados en el proceso (en este directorio)
____rm -rf *.json *.vpp *.vcd
```

Código 1.13: Contenido completo del archivo **Makefile** propuesto para esta sección

### 1.2.6. Reto: Diseñar y simular un restador teniendo como base el sumador de 4 bits

## 1.3. Blink

### 1.3.1. Diseño de un Blink a través de un contador de N bits

**El principio del contador**

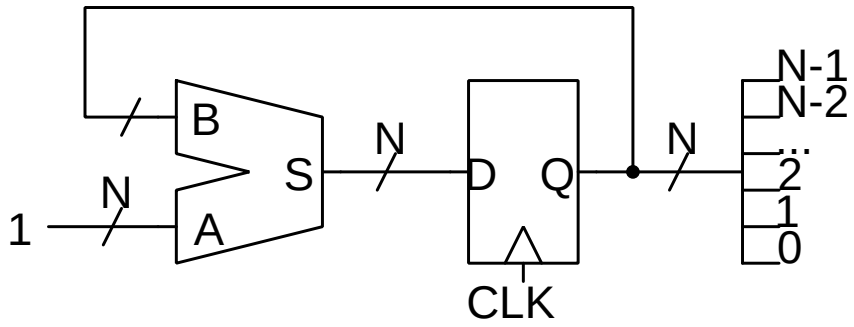



Figura 1.18: Representación en RTL de un contador de N bits

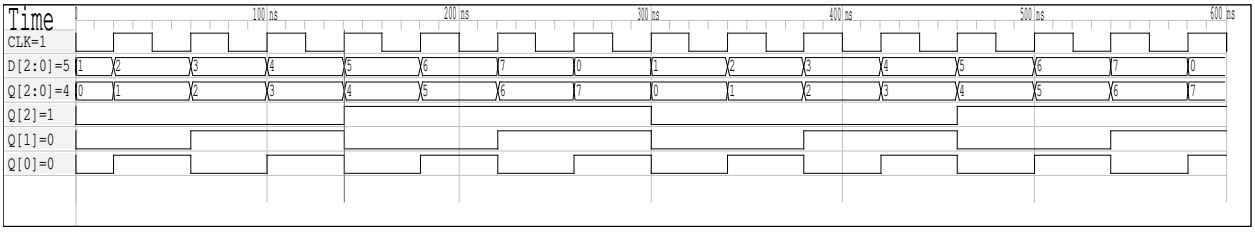



Figura 1.19: Representación en GTKWave de las formas de onda del contador de N bits

### Contador con señal de reinicio

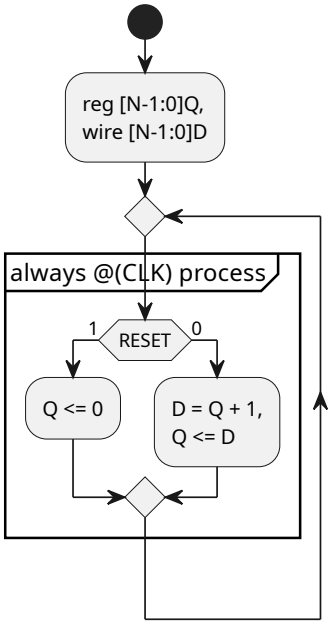



Figura 1.20: Diagrama de flujo del contador de N bits con RESET

```
// filename: counter.v
`ifndef N
`define N 8
`endif
module counter (
    // Inputs and output ports
    input wire CLK,
    input wire RESET,
    output reg [`N-1:0] Q = 0
);
  // Declaración de señales [reg, wire]
  wire [`N-1:0] D;

  // Descripción del comportamiento
  assign D = Q + 1;

  always @(posedge CLK) begin
    if (RESET) begin
      Q <= 0;
    end else begin
      Q <= D;
    end
  end
endmodule
```

Código 1.14: Descripción de un contador de N bits con señal de RESET en *Verilog*

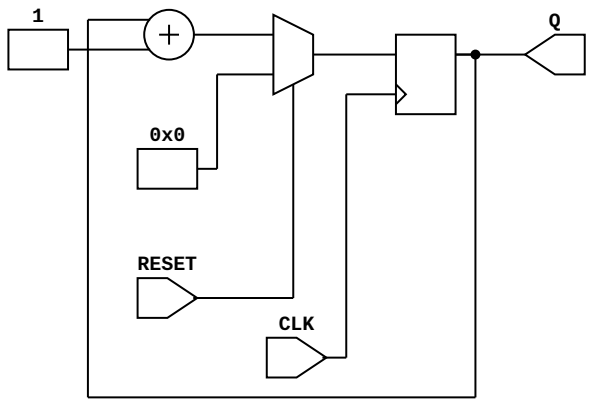



Figura 1.21: Contador con señal de reinicio en RTL

```
// filename: counter.v
`timescale 10ns / 10ns  // <time_unit>/<time_precision
`define N 3
`include "./counter.v"
module counter_tb;

  // Determinar el tamaño del wire de la sonda de prueba
  localparam integer SIZE = `N;

  reg reset = 0;

  // CLOCK STIMULUS
  // Suponiendo que se tiene una señal de reloj de 25 MHz
  // (40 nS como periodo) se puede configurar el clock del testbech
  // para que corresponda a esa escala de tiempo. Teniendo presente
  // que cada tick en este ejemplo es de 10nS (time_unit en timescale) y que cada
  // cambio de estado en el reloj se produce en cada medio ciclo, en cada medio periodo,
  // (20nS), cada cambio en el reloj se debe registrar en cada 2 Ticks de la
  // escala de simulación.
  localparam integer TICKS = 2;
  reg clk = 0;
  always #(TICKS) clk = !clk;
  initial begin
    // Pasado 32 estímuslos finaliza la simulación
    #(32 * TICKS) $finish();  // [stop(), $finish()]
  end
  // RESET STIMULUS
  initial begin
    #0 reset = 0;
    #(20 * TICKS) reset = 1;
    #(2 * TICKS) reset = 0;
  end

  // RESULT FOR DEVICE/DESIGN UNDER TEST
  wire [SIZE-1:0] probe;

  // DEVICE/DESIGN UNDER TEST
  counter dut (
      .CLK(clk),
      .Q(probe),
```

```
    .RESET(reset)
  );

  // WAVES IN VCD TO OPEN IN GTKWAVE
  initial begin
    $dumpvars(0, counter_tb);
  end
endmodule
```

Código 1.15: Testbech para el contador de N bits con ajuste en la escala del tiempo de la simulación

### 1.3.2. Blink con divisor de frecuencia

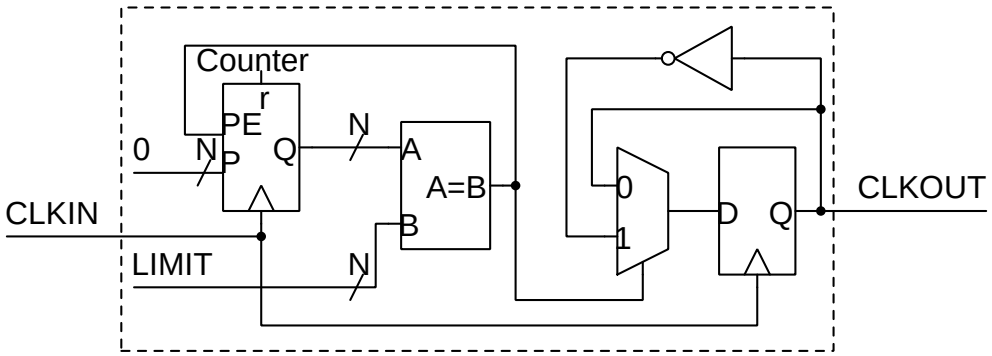



Figura 1.22: Representación en RTL de un divisor de frecuencia

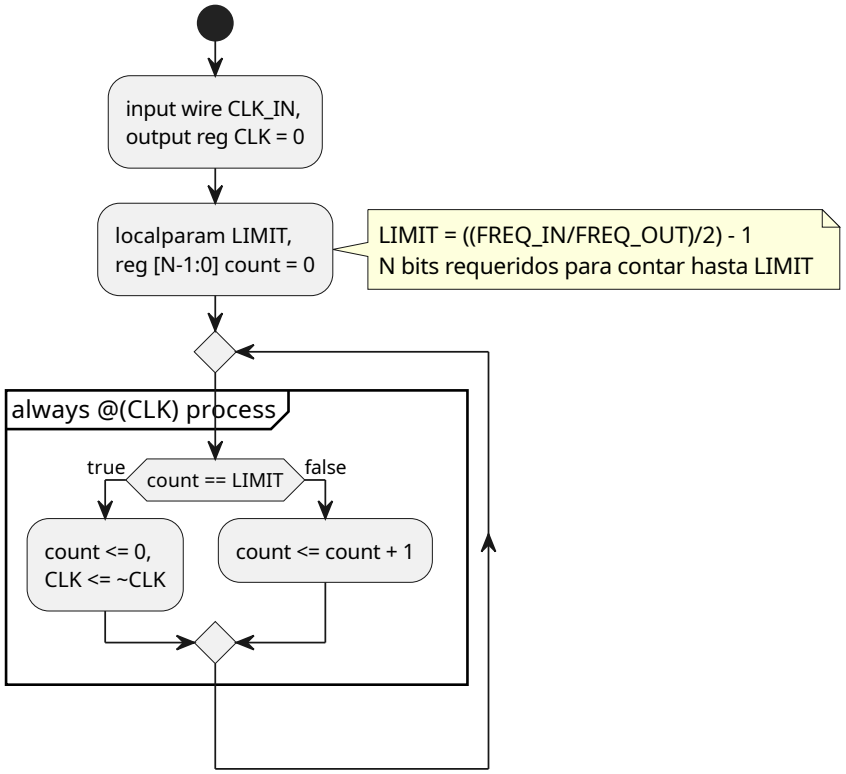



Figura 1.23: Diagrama de flujo para el Blink a partir de un divisor de frecuencia

```
// filename: divFreq.v
`ifndef FREQ_IN
`define FREQ_IN 1000
`endif
`ifndef FREQ_OUT
`define FREQ_OUT 100
`endif
`ifndef INIT
`define INIT 0
`endif

module divFreq (
    // Inputs and output ports
    input CLK_IN,
    output reg CLK_OUT = 0
);

  localparam integer COUNT = (`FREQ_IN / `FREQ_OUT) / 2;
  localparam integer SIZE = $clog2(COUNT);
  localparam integer LIMIT = COUNT - 1;

  // Declaración de señales [reg, wire]
  reg [SIZE-1:0] count = `INIT;

  // Descripción del comportamiento
  always @(posedge CLK_IN) begin
    if (count == LIMIT) begin
      count   <= 0;
```

```
    CLK_OUT <= ~CLK_OUT;
  end else begin
    count <= count + 1;
  end
 end
endmodule
```

Código 1.16: Divisor de frecuencia descrito en verilog

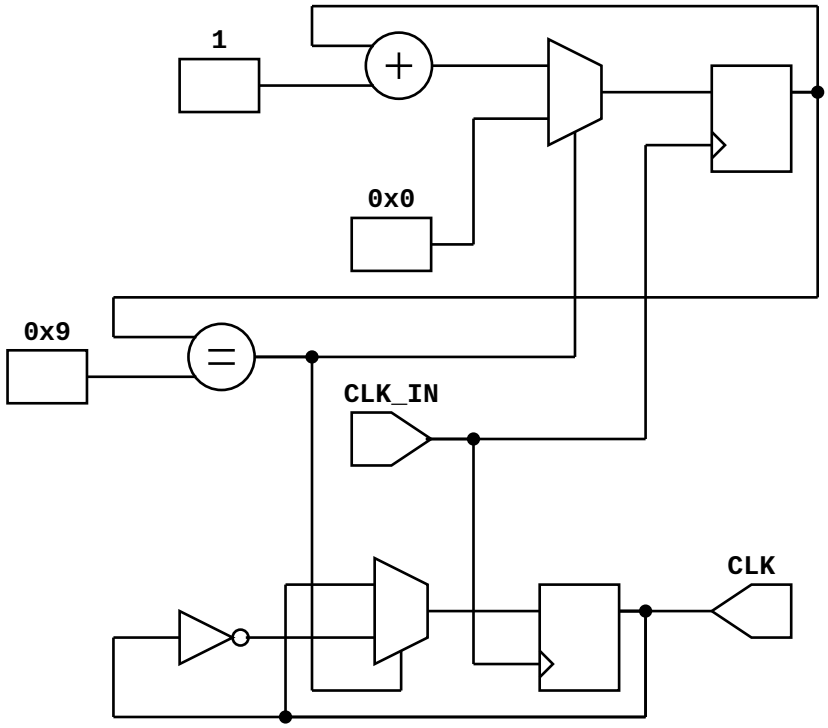



Figura 1.24: Diagrama RTL del divisor de frecuencia

```
// filename divFreq_tb.v
`include "./divFreq.v"

`define TICKS ((`FREQ_IN / `FREQ_OUT) / 2) * 4
`define TIME_UNIT (1 / `FREQ_IN)
// `timescale 10ns / 10ns
module divFreq_tb;

 // Determinar el tamaño de los wire como de los estímulos
 parameter integer OUTPUT_SIZE = 1;

 // CLOCK STIMULUS
 // Make a regular pulsing clock.
 reg clk = 0;
 always #1 clk = !clk;

 initial begin
  #(`TICKS) $finish();  // [stop(), $finish()]
 end
```

```
  // RESULT FOR DEVICE/DESIGN UNDER TEST
  wire [OUTPUT_SIZE-1:0] probe;

  // DEVICE/DESIGN UNDER TEST
  divFreq dut (
      .CLK_IN (clk),
      .CLK_OUT(probe[0])
  );

  // WAVES IN VCD TO OPEN IN GTKWAVE
  initial begin
    $dumpvars(0, divFreq_tb);
  end
endmodule
```

Código 1.17: Testbech para el divisor de frecuencia de N bits con ajuste en la escala del tiempo de la simulación

```
# filename: Makefile
# Brief: Makefile para simulación, sintesis, configuración y generación de RTL con la BlackIce40
# Identificador v para representar los archivos verilog
v?=./divFreq.v
# Identificador top para representar el top del RTL a visualizar
TOP_NAME=$(basename $(notdir $v))
top?=$(TOP_NAME)
# Identificador tb para el archivo verilog que contiene el testbench
tb?=$(TOP_NAME)_tb.v
# El identificador "MACRO_SIM" permite ajustar los valores de las diferentes definiciones del diseño, ej:
# MACRO_SIM=-DFREQ_IN=100 -DFREQ_OUT=50
MACRO_SIM=
# Puerto de comunicación SERIAL para la programación de la BlackIce40.  En caso de no detectar el
# puerto serial, listar con ls /dev/tty* y poner el puerto correspondiente, por default está ttyACM0.
SERIAL=/dev/ttyACM0

sim:
	# 1. Crear el archivo .vvp ejecutable desde iverilog
	iverilog $(MACRO_SIM) -o $(tb).vvp $(tb)
	# 2. Ejecuta el archivo .vvp para mostrar resultados
	vvp $(tb).vvp -dumpfile=$(top).vcd

wave:
	gtkwave $(top).vcd $(top).gtkw

rtl:
	# 1. Síntesis del diseño, si la sintaxis es correcta, se genera un archivo json que representa el diseño
	yosys $(MACRO_SIM) -p 'read_verilog $v; prep -top $(top); hierarchy -check; proc; write_json $(top).json'
	# 2. Comando para generación del RTL en formato SVG (vectorial)
	netlistsvg $(top).json -o $(top).svg
	# 3. Visualizar el RTL con el visor de imagenes eog
	eog $(top).svg

syn:
	# 1. Sintesis
	yosys -p "synth_ice40 -top $(top) -json $(top).json" $v
	# 2. Place and route
	nextpnr-ice40 --hx4k --package tq144 --json $(top).json --pcf $(top).pcf --asc $(top).pnr
	# 3. Empaquetar en un bitstream
	icepack $(top).pnr $(top).bin
```

```
config:
	stty -F $(SERIAL) raw
	cat $(top).bin > $(SERIAL)

clean:
	rm -f *.vvp *.json *.vcd *.pnr *.bin
```

Código 1.18: Makefile con definición de macros para modificar los valores del diseño

## 1.4. PWM

La modulación por ancho de pulso (PWM) es usada en ciertos dispositivos, como es el caso de servomotores, drivers para control de velocidad de motores, entre otros.

La modulación por ancho de pulso consiste en una señal digital periódica, a la cual, se le modifica el tiempo que dura en el estado lógico 1. Mientras que se encuentra en este estado lógico, se considera que es útil para realizar un trabajo. A este tiempo se le denomina *Duty Cicle* y es representado de manera porcentual. Por ejemplo, a una señal PWM con un periodo de un 1 milisegundo donde $0{,}5mS$ está en alto (1 lógico) y la otra parte del periodo ($0{,}5mS$) en estado bajo (cero lógico), se determina que tiene un ciclo útil de 50 %.

Para diseñar un periférico PWM se requiere controlar el periodo de la señal, como también indicar el ciclo útil que tendrá. En la imagen 1.25 se puede observar, un contador al cual se le indica un límite de conteo (LIMIT) que está asociado directamente al periodo de la señal, también, se cuenta con un circuito comparador que va observando el valor del conteo junto a un valor de referencia (DUTY); este comparador genera una señal de salida con un 1 lógico si el contador aún no ha superado el valor de referencia. Si el valor de referencia no varía, la señal PWM se mantendrá constante, es decir, con su mismo ciclo útil

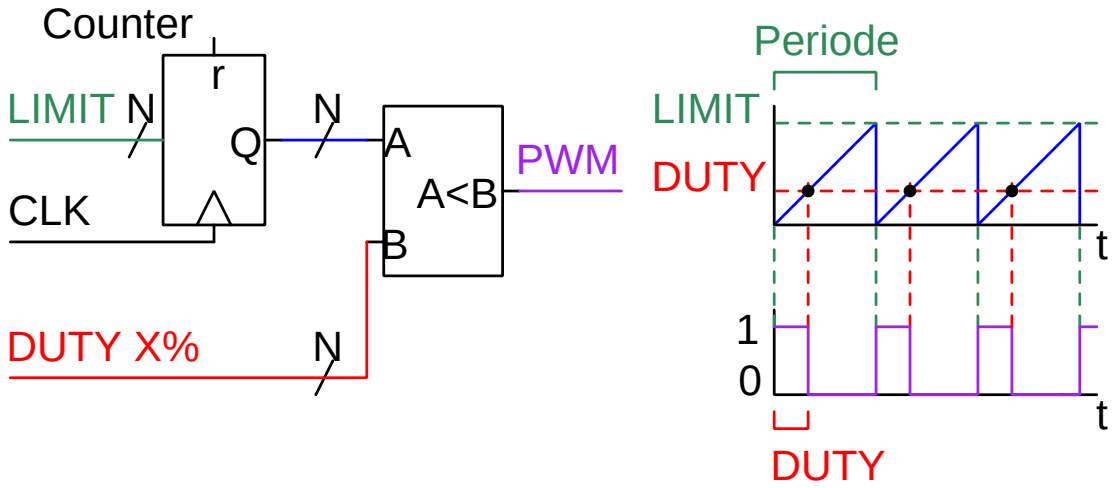



Figura 1.25: Circuito digital de un modulador por ancho de pulso (PWM)

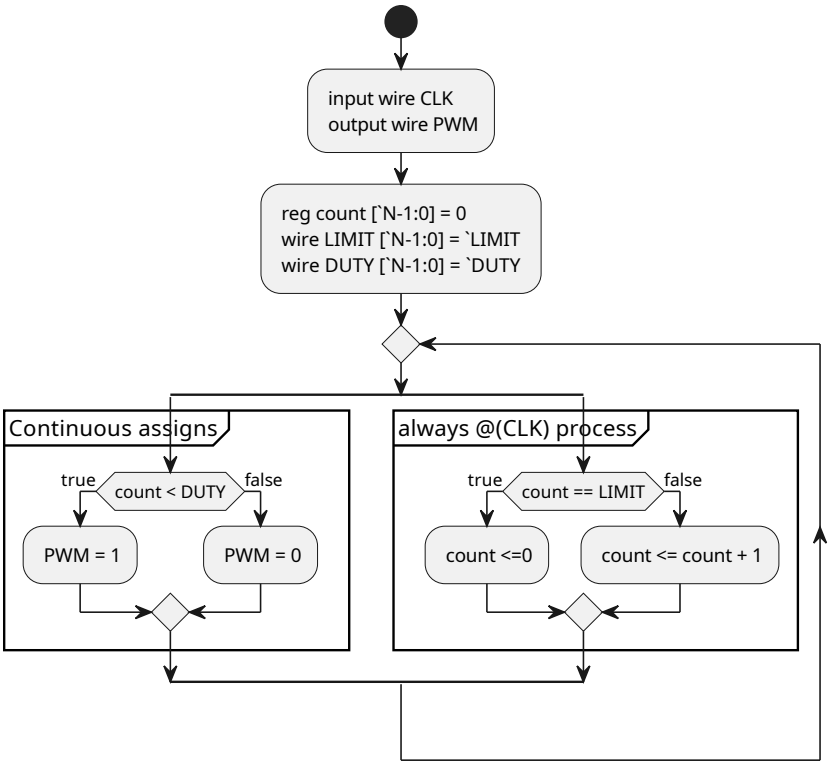



Figura 1.26: Diagrama de flujo del PWM

# Apéndice A

# Instalación de herramientas

## A.1. Instalación de Conda

Miniconda es un ecosistema para homogeneizar las características requeridas para la instalación y ejecución de aplicaciones, a través de variables de entorno y un gestor de paquetes online, lo anterior facilita la instalación de aplicaciones con un solo comando y evita los problemas que se asocien a estos mismos.

Instalación de dependencias

```
$ sudo apt update
$ sudo apt install wget curl eog
```

Ejecute los siguientes 3 comandos en una terminal de Linux, si tiene algún error comparta el resultado en foros para recibir sugerencias.

Instalación de miniconda

```
$ cd Downloads # o en Descargas si tiene su S.O. en español
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash Miniconda3-latest-Linux-x86_64.sh
```

## A.2. Instalación de herramientas de diseño en Conda

*Conda* cuenta con un gestor paquetes en Internet llamado *Anaconda* [1] el cual ofrece paquetes de librerías y aplicaciones compilados para diferentes arquitecturas de computadores. Para este caso, se requiere la instalación de herramientas opensource que hacen parte del flujo de diseño de sistemas digitales, las cuales son listadas a continuación:

[1] Sitio oficial de anaconda `https://anaconda.org/`

Lista de paquetes conda a instalar para el diseño de sistemas digitales

Entorno conda con python 3.7
conda-forge::libstdcxx-ng
conda-forge::libusb
conda-forge::libftdi
conda-forge::libhidapi
johnnycubides::openfpgaloader
litex-hub::nextpnr-ice40
litex-hub::nextpnr-ecp5
litex-hub::iceprog
litex-hub::yosys
litex-hub::iverilog
symbiflow::netlistsvg
conda-forge::graphviz
conda-forge::gtkwave
conda-forge::verilator
litex-hub::gcc-riscv64-elf-newlib

Para instalar las herramientas de diseño de sistemas digitales opensource se recomienda ejecutar el siguiente comando, el cual realiza el proceso automático de descarga de paquetes desde el gestor de paquetes *Anaconda*, la instalación de los mismos y la configuración de la variable de entorno que por conveniencia se llamará **digital**. El valor agregado de hacer uso del comando se basa en que la instalación que se realiza es una cópia de la configuración realizada en un equipo con las herramientas probadas para los diferentes procesos. Lo anterior evita problemas asociados a la instalación de diferentes versiones de las librerías y o aplicaciones que puedan generar errores al momento de ejecutar un diseño o un ejemplo de este libro.

Instalación de herramientas de diseño para sistemas digitales en *Conda*

```
$ curl https://raw.githubusercontent.com/johnnycubides/\
digital-electronic-1-101/main/installTools/spec-file.txt\
> ./spec-file.txt && conda create -n digital --file ./spec-file.txt
```

### A.2.1. Permisos de hardware

Para realizar funciones de configuración de sistema digital en una FPGA o compartir información con la FPGA en algún protocolo de hardware, Linux, se requiere dar permisos de uso de hardware, para facilitar el proceso puede ejecutar el siguiente comando:

Dar permisos de uso del hardware en Linux

```
$ curl https://raw.githubusercontent.com/johnnycubides/\
digital-electronic-1-101/main/installTools/hw-permissions.sh | sh
```

### A.2.2. Iniciando la variable de entorno de digital en Conda

Para que las herramientas y librerías para el flujo de diseño estén disponibles, se requiere activar la variable de entorno que se ha nombrado como **digital** en Conda. Para tal finalidad podrá ejecutar el siguiente comando en una *Shell*:

Inicio de variable de entorno **digital** para acceder a las herramientas de diseño digital

Ejecute el siguiente comando en una *Shell*

```
$ conda activate digital
```

Lo anterior dará como resultado en el prompt la indicación de la variable de entorno de digital activada como en el siguiente ejemplo:
**(digital) $**

El anterior comando con su respuesta indica que para esa *Shell* estará disponibles las herramientas instaladas en la variable de entorno **digital** de Conda. Si por ejemplo, abre otra *Shell* o la reinicia, notará que desaparece en el prompt la indicación de la variable de entorno y por tanto deberá activar de nuevo el entorno, reescribiendo el anterior comando.

Si esto le causa problemas debido a que debe activar la variable de entorno cada vez que reinicia una *Shell* **puede considerar** indicarle en la configuración de la *Shell* que inicie la variable de entorno como se indica a continuación:

Agregar la activación de la variable **digital** en el arranque de la *Shell*

```
$ echo "conda activate digital" >> ~/.bashrc
```

Al reiniciar la *Shell*, deberá aparecer en el prompt el símbolo $ antecedido por la variable de entorno de conda llamada digital, ejemplo:
**(digital) $**

Si desea que no se inicie automáticamente la variable de entorno de **digital** cada vez que abre una *Shell* podrá editar el archivo **.bashrc** (que se encuentra en el directorio home de su usuario) borrando la línea correspondiente a **conda activate digital** que estará al final del archivo (si no ha realizado más modificaciones en el archivo) y bastará con reiniciar la *Shell*.

## A.3. *Digital*: Entorno de diseño y simulación de circuitos digitales

*Digital* [2] es una herramienta de diseño y simulación de circuitos digitales *opensource* creada por Helmut Neemann. Esta herramienta está escrita en Java lo que indica que se requiere tener instalado la máquina virtual de Java para poder ejecutar la aplicación. Puede verificar si cuenta con una versión de Java compatible con *Digital*, en caso de no tenerla, puede instalar la versión del *openjdk-11-jdk* para una distribución de Linux basada en *Debian* como se ve en el siguiente cuadro, en el caso de tener otra distribución deberá buscar en foros como realizar la instalación.

Instalación de dependencias requeridas para *Digital* desde *Shell*

```
$ sudo apt install openjdk-11-jdk
```

Los siguientes comandos de instalación de *Digital* se pueden ejecutar en cualquier distribución Linux, si tiene problemas con alguno de los comandos consulte los foros.

[2] Repositorio de Digital: `https://github.com/hneemann/Digital`

Comando de instalación de *Digital* desde la línea de comandos *Shell*

1. Creación de un directorio donde se almacene la aplicación portable

```
$ mkdir -p ~/gitPackages/ && cd ~/gitPackages
```

2. Descarga de la aplicación *Digital* en el directorio creado.

```
$ wget https://github.com/hneemann/Digital/releases/latest/download/Digital.zip
```

3. Desempaqutando aplicación.

```
$ unzip Digital.zip
```

4. Entrando al directorio de *Digital* para iniciar el proceso de enlazado

```
$ cd Digital/
```

5. Instalación de *Digital* en el menú de aplicaciones

```
$ ./install.sh
```

6. Creación de enlace simbólico para lanzar la aplicación desde el *Shell*

```
$ sudo ln -sr ./Digital.sh /usr/local/bin/
```

Para lanzar o iniciar *Digital* lo puede hacer de dos maneras; en primer lugar, puede buscar en el menú de aplicaciones con el nombre de Digital, donde aparecerá el ícono de la aplicación con el nombre y posiblemente en la categoría de educación. En segundo lugar, lo puede lanzar desde la *Shell* ejecutando el comando como se ve en el siguiente cuadro. Cabe resaltar que si va a hacer uso de *IVerilog* como complemento para las simulaciones, deberá activar el entorno de digital de *conda* que anteriormente se instaló, recuerde que la instrucción de activación del entorno digital se realiza en el *Shell* con el comando: **conda activate digital**.

Lanzamiento de aplicación *Digital* desde *Shell*

Puede abrir una terminal y ejecutar el siguiente comando para iniciar *Digital*

```
$ Digital.sh
```

## A.4. Geany: Light IDE

Instalación de Geany

```
$ sudo apt update
```

```
$ sudo apt install geany geany-plugins
```

## A.5. Pulseview: Analizador Lógico

# Apéndice B

# Flujo de diseño, makefiles y pinout de tarjetas de desarrollo

## B.1. BlackIce iCE40-HX4K

### B.1.1. Flujo de diseño para placa de desarrollo BlackIce

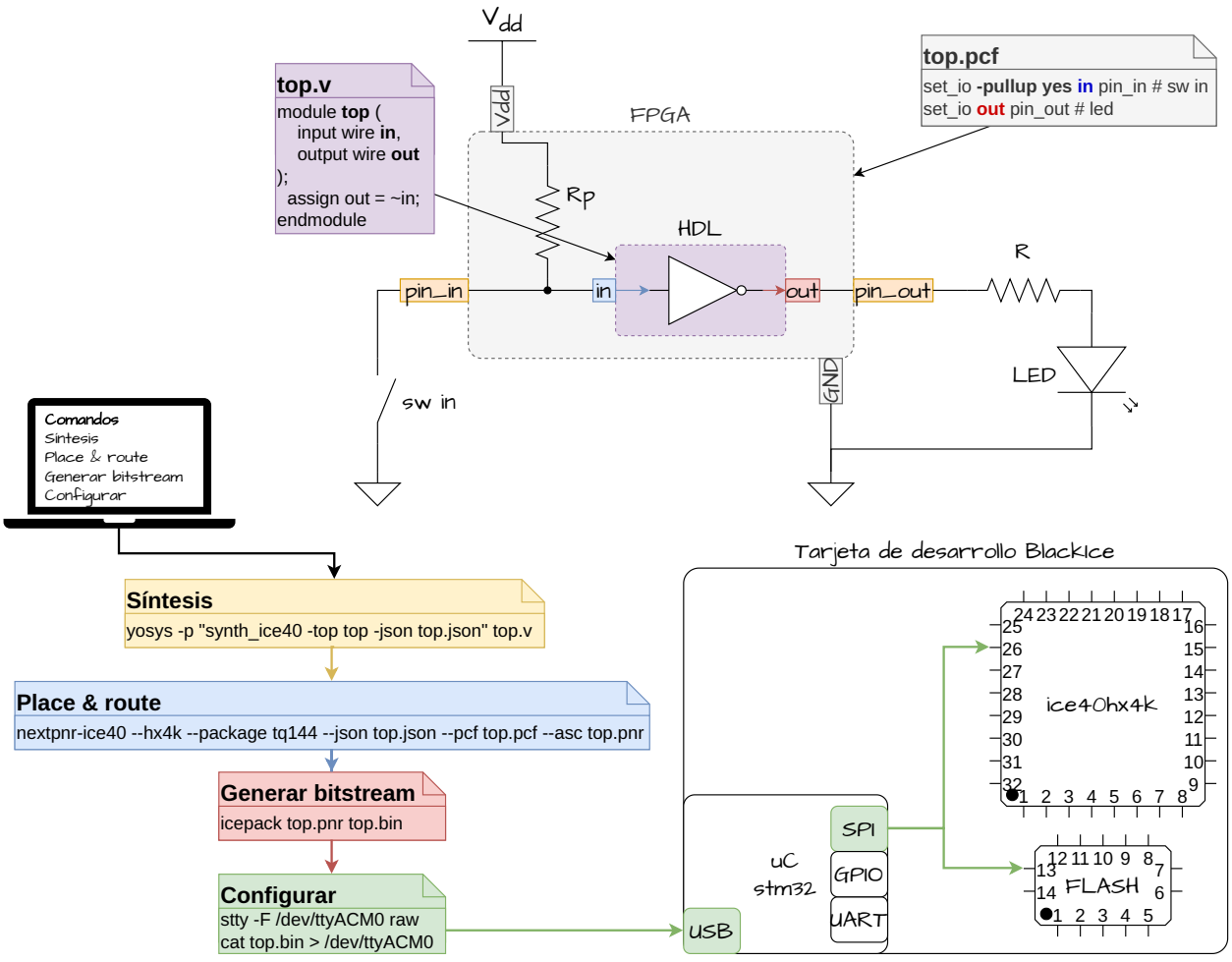



Figura B.1: Flujo de diseño de circuito digital tarjeta de desarrollo BlackIce

### B.1.2. Pinout tarjeta de desarrollo BlackIce

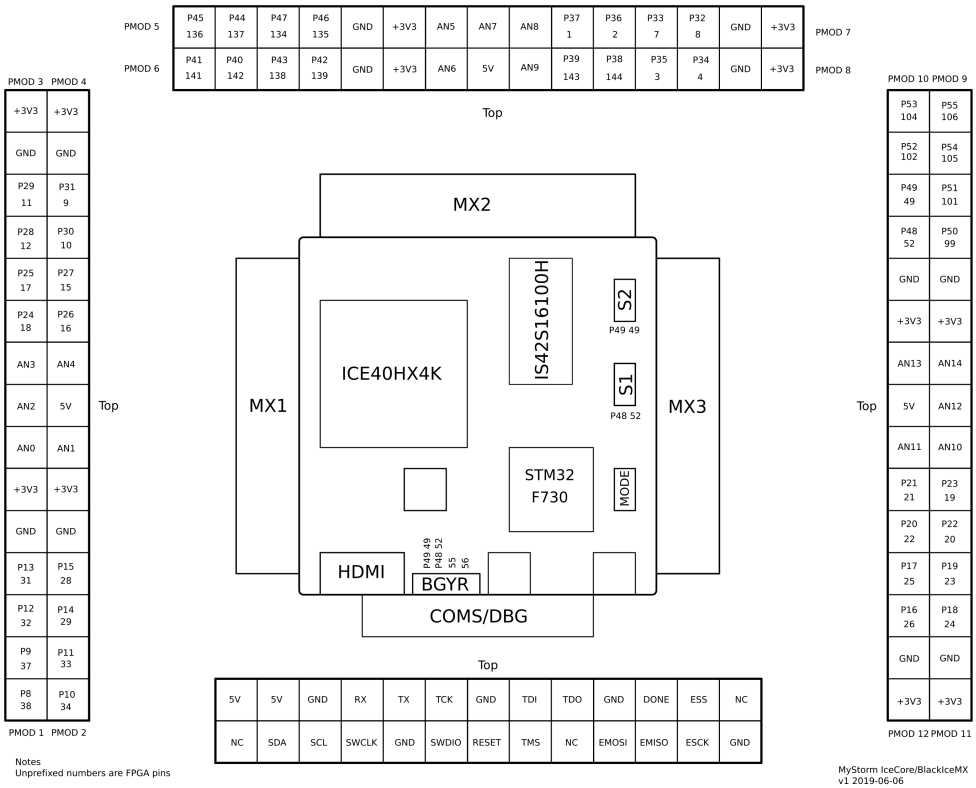



Figura B.2: Pinout tarjeta de desarrollo BlackIce

### B.1.3. Makefile para tarjeta de desarrollo BlackIce

```
# filename: Makefile
# Brief: Makefile para simulación, sintesis, configuración y generación de RTL con la BlackIce40
# Identificador v para representar los archivos verilog
v?=top.v
# Identificador top para representar el top del RTL a visualizar
TOP_NAME=$(basename $(notdir $v))
top?=$(TOP_NAME)
# Identificador tb para el archivo verilog que contiene el testbench
tb?=$(TOP_NAME)_tb.v
# El identificador "MACRO_SIM" permite ajustar los valores de las diferentes definiciones del diseño, ej:
# MACRO_SIM=-DFREQ_IN=100 -DFREQ_OUT=50
MACRO_SIM=
# Puerto de comunicación SERIAL para la programación de la BlackIce40.  En caso de no detectar el
# puerto serial, listar con ls /dev/tty* y poner el puerto correspondiente, por default está ttyACM0.
SERIAL=/dev/ttyACM0

sim:
	# 1. Crear el archivo .vvp ejecutable desde iverilog
	iverilog $(MACRO_SIM) -o $(tb).vvp $(tb)
```

```
__# 2. Ejecuta el archivo .vvp para mostrar resultados
__vvp $(tb).vvp -dumpfile=$(top).vcd

wave:
__gtkwave $(top).vcd $(top).gtkw

rtl:
__# 1. Síntesis del diseño, si la sintaxis es correcta, se genera un archivo json que representa el diseño
__yosys $(MACRO_SIM) -p 'read_verilog $v; prep -top $(top); hierarchy -check; proc; write_json $(top).json'
__# 2. Comando para generación del RTL en formato SVG (vectorial)
__netlistsvg $(top).json -o $(top).svg
__# 3. Visualizar el RTL con el visor de imagenes eog
__eog $(top).svg

syn:
__# 1. Sintesis
__yosys -p "synth_ice40 -top $(top) -json $(top).json" $v
__# 2. Place and route
__nextpnr-ice40 --hx4k --package tq144 --json $(top).json --pcf $(top).pcf --asc $(top).pnr
__# 3. Empaquetar en un bitstream
__icepack $(top).pnr $(top).bin

config:
__stty -F $(SERIAL) raw
__cat $(top).bin > $(SERIAL)

clean:
__rm -f *.vvp *.json *.vcd *.pnr *.bin
```

Código B.1: Makefile para simulación, síntesis, configuración y generación de RTL para tarjeta de desarrollo BlackIce

## B.2. Colorlight ECP5

### B.2.1. Flujo de diseño para placa de desarrollo Colorlight

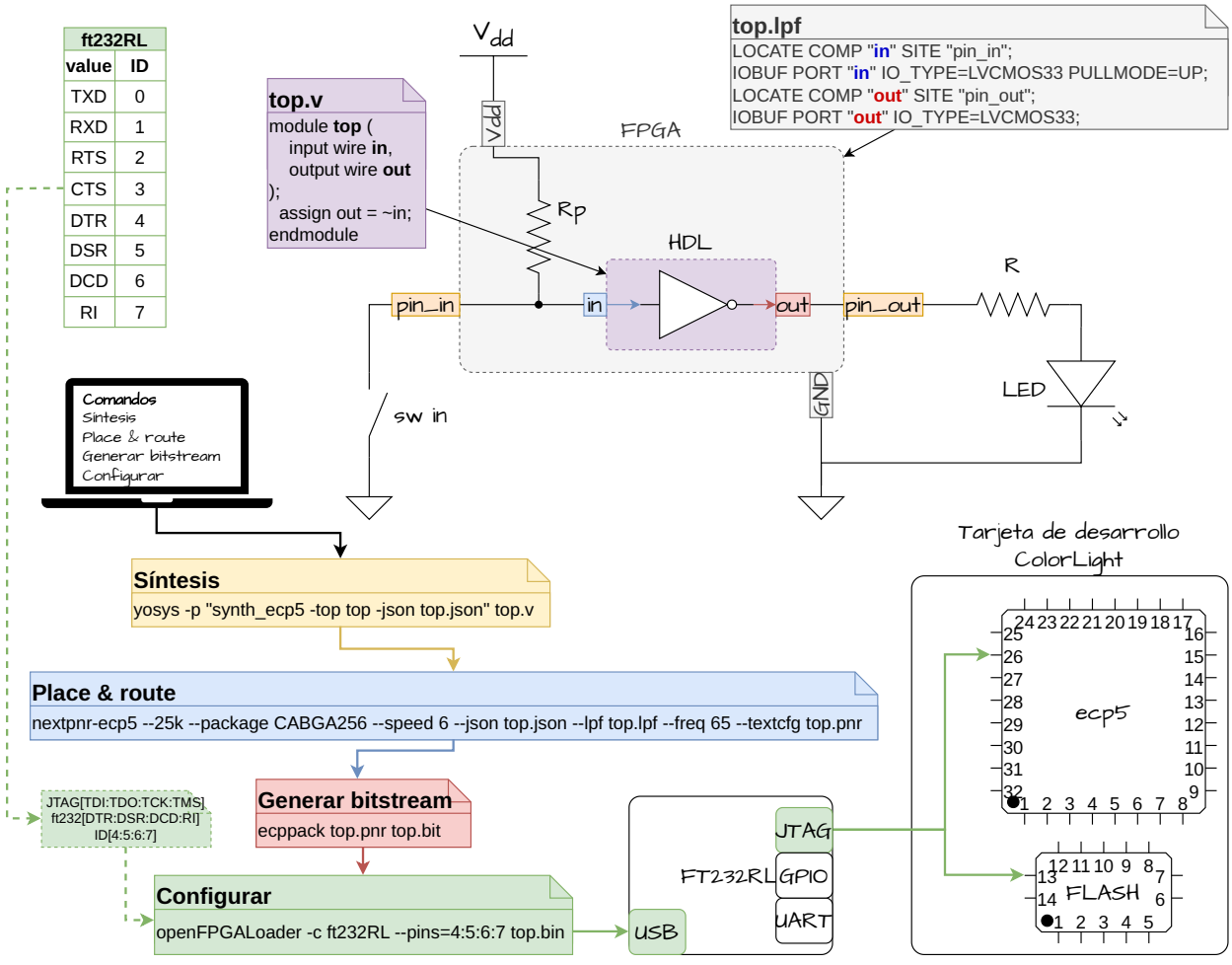



Figura B.3: Flujo de diseño de circuito digital tarjeta de desarrollo Colorlight

### B.2.2. Pinout tarjeta de desarrollo Colorlight 5A-75E V7.1

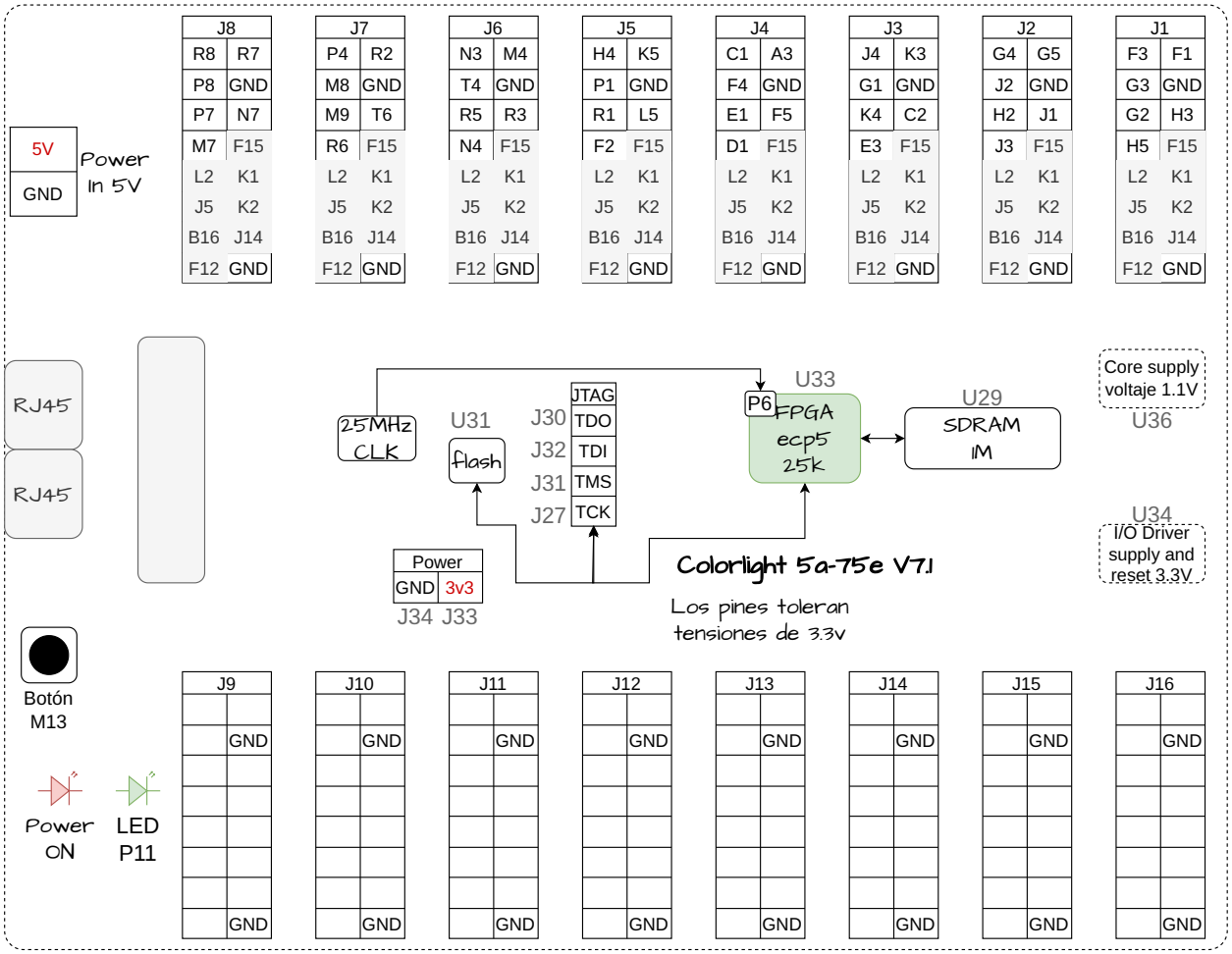



Figura B.4: Pinout de tarjeta de desarrollo Colorlight 5A-75E V7.1

### B.2.3. Pinout tarjeta de desarrollo Colorlight 5A-75E V8.2

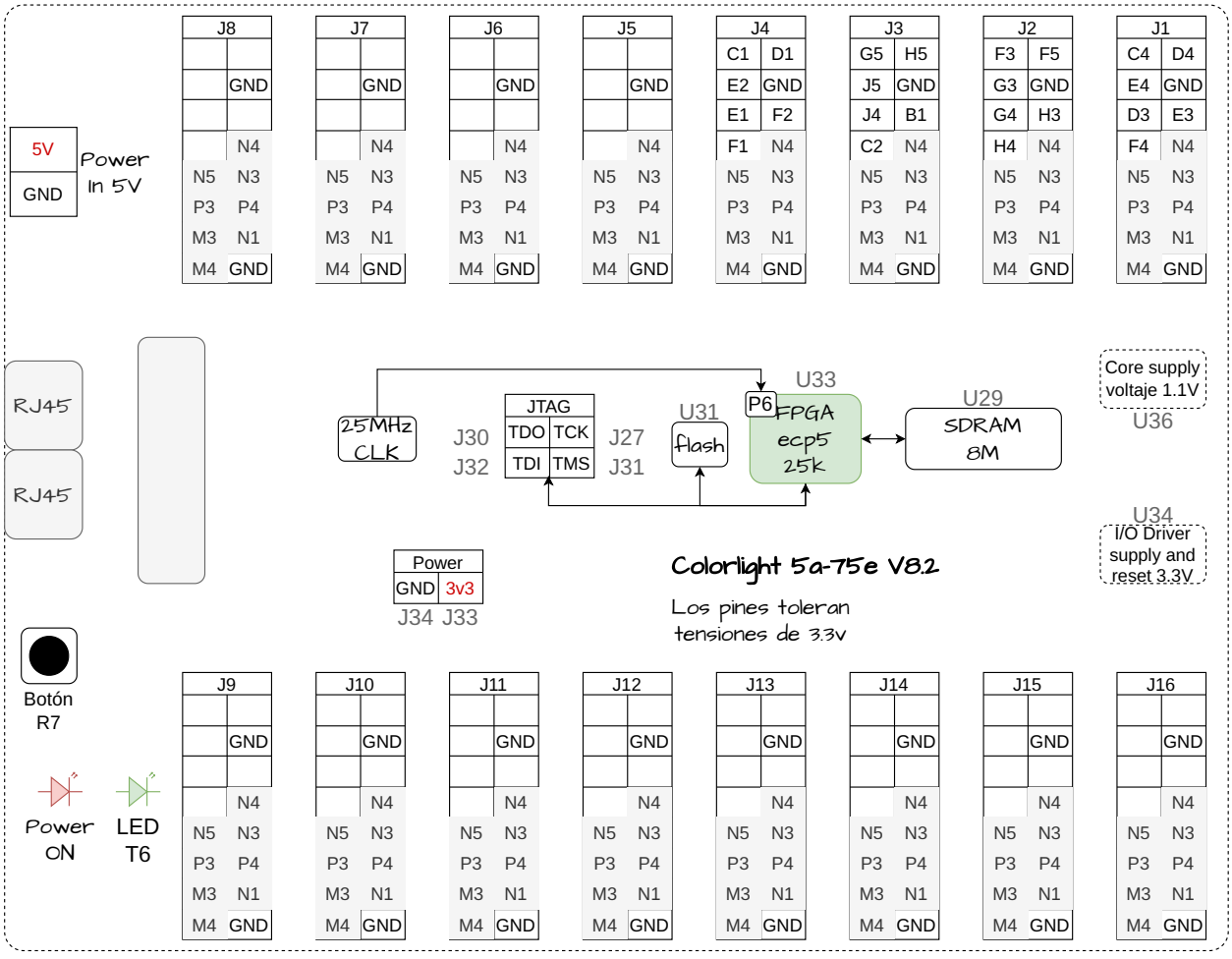



Figura B.5: Pinout de tarjeta de desarrollo Colorlight 5A-75E V8.2

## B.3. iCE40-HX8K Breakout Board

### B.3.1. Flujo de diseño para placa de desarrollo iCE40-HX8K

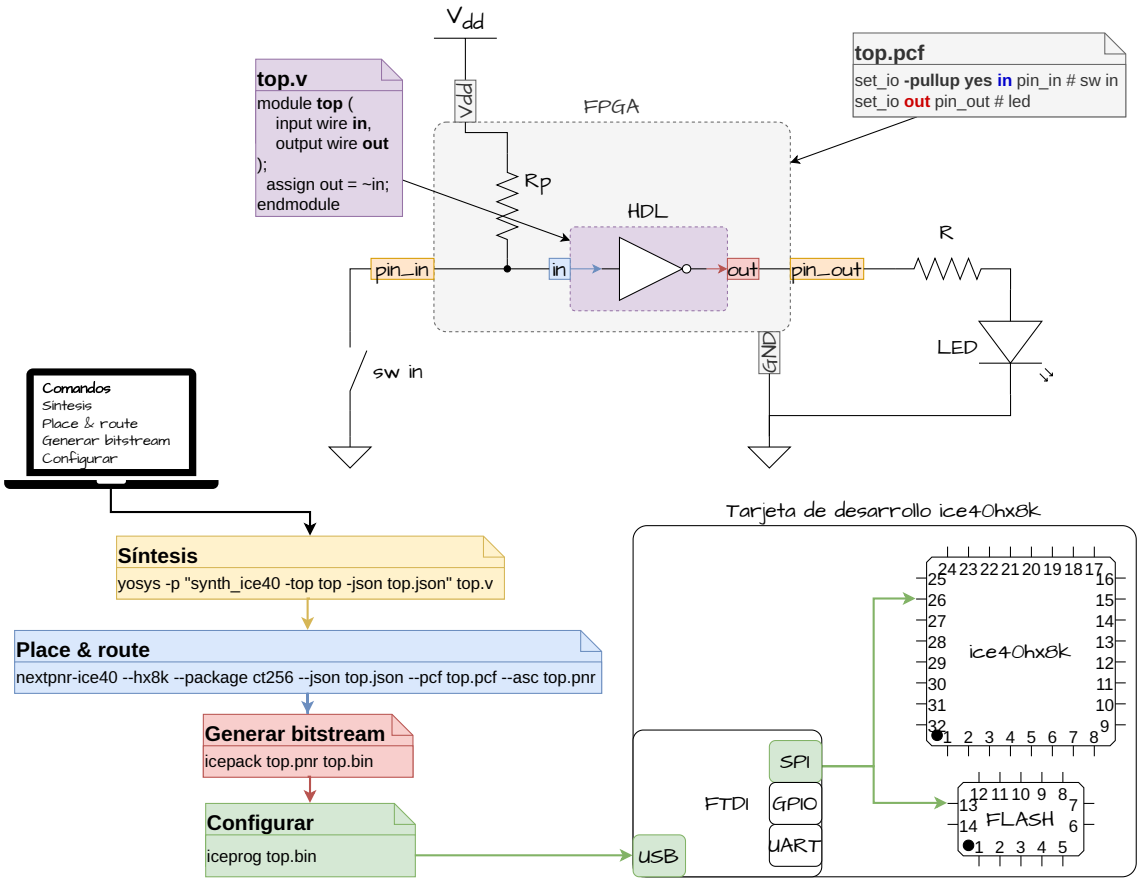



Figura B.6: Flujo de diseño de circuito digital para tarjeta de desarrollo iCE40-HX8K