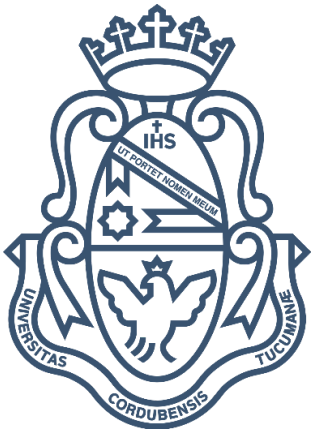


Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales;

Arquitectura de Computadoras

Trabajo Práctico No 2: Módulo UART

Profesores :

Pereyra, Martín

Alonso, Martín

Ciordia Cantarella, Francisco

Alumnos (por órden alfabético):

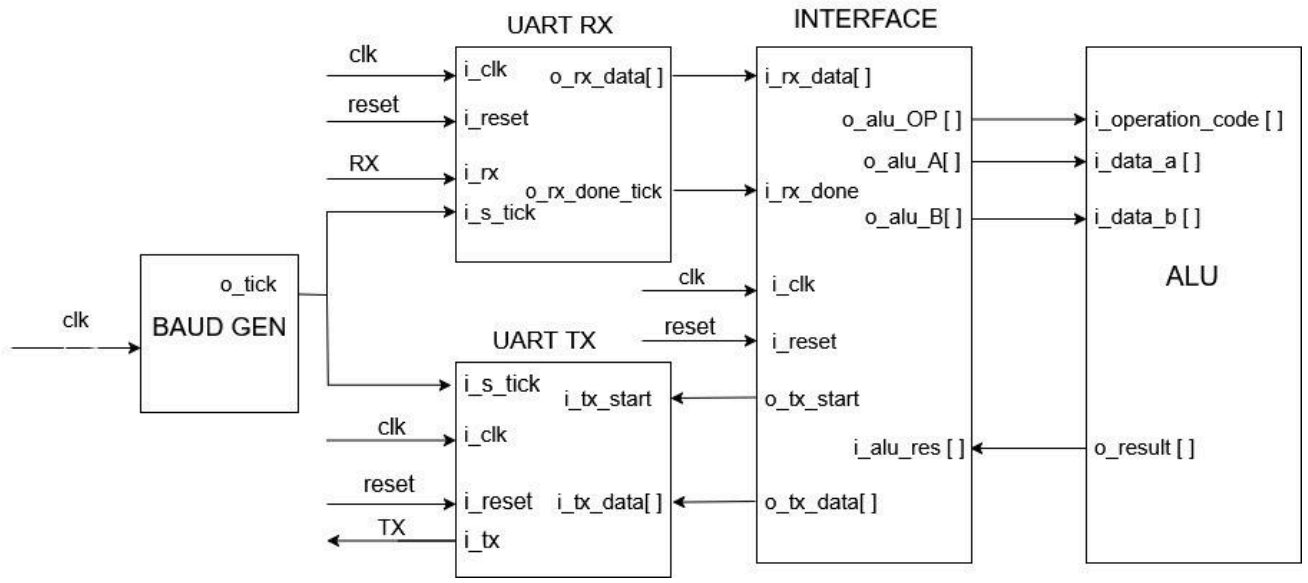
Badariotti, Juan miguel

Erlicher, Ezequiel

Descripción

Este trabajo consistió en diseñar e implementar 2 módulos UART(uno transmisor y otro receptor) los cuales interactúan con la ALU desarrollada en el trabajo práctico 1 mediante una interfaz.Los módulos UART están sincronizados mediante un generador de Baudios que genera un “tick” cada 163 ciclos de reloj para generar una frecuencia de muestreo de 307.200 ticks por segundos. Estos parámetros de funcionamiento están pensados para un clock de sistema con una frecuencia de 50MHz.

El diagrama lógico del sistema UART es el siguiente:



Estructura del código

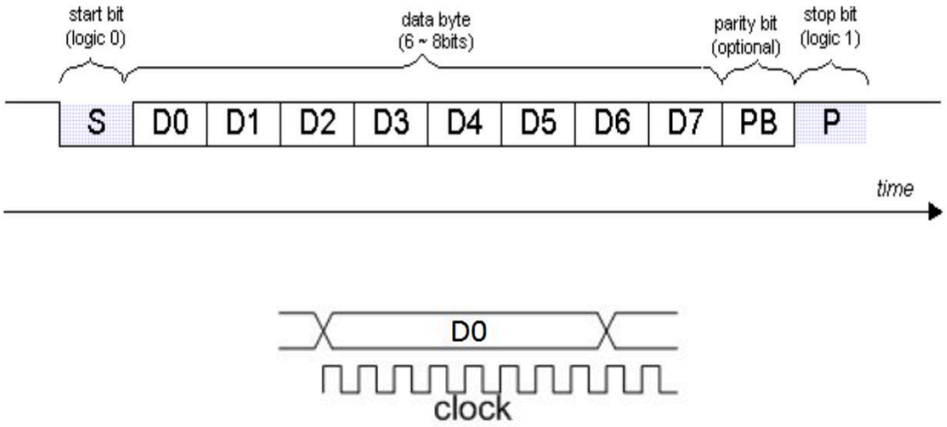
baud_gen.v

Este módulo implementa un generador de baudios (unidad de medida que representa el número de símbolos por segundo en un medio de transmisión). El propósito de este generador es generar pulsos `o_tick` utilizados para controlar la velocidad de transmisión y recepción del sistema UART. Recibe como entradas una señal de reloj del sistema `i_clk` y una señal de reseteo `i_reset`. Basándose en los parámetros especificados (la frecuencia del reloj `CLOCK_FREQ` y la velocidad de transmisión deseada `BAUD_RATE`) calcula un factor de división MOD que determina cuántos ciclos de reloj corresponden a un tick a 16 veces la velocidad de transmisión (para sobremuestreo UART). El contador interno (`r_counter`) se incrementa con cada pulso de reloj hasta alcanzar `MOD-1`, momento en el que se reinicia a cero y se genera un pulso de un solo ciclo (`o_tick`).

```
module baud_gen
  #(
        parameter  BAUD_RATE  = 19200,
        parameter  CLOCK_FREQ = 50_000_000,
        parameter  NB_COUNTER = 16
    )
    (
        input  wire    i_clk,
        input  wire    i_reset,
        output wire    o_tick
    );

    localparam integer MOD = (CLOCK_FREQ + (BAUD_RATE * 16) - 1) / (BAUD_RATE * 16);

    reg  [NB_COUNTER-1:0]  r_counter;
    wire [NB_COUNTER-1:0]  r_next;

  always @ (posedge i_clk)begin
    if (i_reset)
            r_counter <= 0;
        else
            r_counter <= r_next;
    end

    // Alcanzó el valor máximo del contador?
    wire last_value = (r_counter == MOD-1);
    // Siguiente valor del contador
    assign r_next = last_value ? 0 : r_counter + 1;
    // Tick pulse
    assign o_tick = last_value ? 1'b1 : 1'b0;

endmodule
```

rx_mod

Este módulo implementa el receptor UART. El diseño se basa en una máquina de estados finitos (FSM) que controla las diferentes etapas de recepción. Su función es tomar los bits que ingresan en serie a través de su entrada serial y reconstruir el byte de datos correspondiente. El receptor espera por el bit de inicio,almacena los bits de datos subsiguientes, verifica el bit de stop y emite el byte recibido junto con una señal que indica la recepción de una trama válida. Cada trama que se recibe tiene la siguiente estructura:



El módulo comienza definiendo dos parámetros: `NB_DATA`, que determina cuántos bits componen una palabra recibida (8 por defecto), y `STOP_TICKS`, que establece el número de ticks del generador de baudios que definen la duración del bit de stop (16 por defecto). Las entradas incluyen el reloj del sistema `i_clk`, la señal del generador de baudios `i_s_tick` que controla la temporización, la línea de entrada serie por donde se reciben los bits `i_rx` y una señal de reinicio `i_reset`. Las salidas son `o_rx_data`, que proporciona el byte de datos recibido, y `o_rx_done_tick`, una señal que indica cuándo se ha recibido y almacenado una trama completa.

La máquina de estados finitos (FSM) tiene cuatro estados, representados por parámetros locales:

- RX_IDLE_STATE
- RX_START_STATE
- RX_DATA_STATE
- RX_STOP_STATE

Estos corresponden respectivamente a la espera del receptor, la orden para comenzar una nueva recepción indicada por el bit de start, la recepción de los bits de datos y la confirmación del bit de stop.

```
module rx_mod
#(

    parameter NB_DATA=8,
    parameter STOP_TICKS = 16
)
(
    input wire i_clk,
    input wire i_s_tick,
    input wire i_rx,
    input wire i_reset,

    output wire [NB_DATA-1:0] o_rx_data,
    output wire o_rx_done_tick
);


// Estados
localparam RX_IDLE_STATE= 2'b00;
localparam RX_START_STATE= 2'b01;
localparam RX_DATA_STATE=2'b10;
localparam RX_STOP_STATE=2'b11;
```

El diseño declara varios registros para controlar el funcionamiento del receptor.

- `rx_state` y `next_rx_state` almacenan los estados actual y siguiente de la FSM.
- `data_counter` y `next_data_counter` registran cuántos bits de datos se han recibido.
- `ticks_counter` y `next_ticks_counter` llevan el recuento de los ticks del generador de baudios
- `data` y `next_data` almacenan los bits que se van recibiendo
- `rx_done` es un bit que se activa durante un ciclo de reloj cuando se ha recibido correctamente una trama completa.

El primer bloque `always` secuencial, activado por los flancos ascendentes de `` `i_clk` `` o `` `i_reset` ``, actualiza todos estos registros. Al reiniciarse, la FSM se inicia en el estado inactivo, se borran todos los contadores y el registro de datos se pone a cero. En caso contrario, el estado actual y las variables se actualizan con sus valores siguientes.

```
// Registros de estado,contador de data,contador de ticks y data
reg[1:0] rx_state,next_rx_state;
reg[2:0] data_counter,next_data_counter;
reg[3:0] ticks_counter,next_ticks_counter;
reg[NB_DATA-1:0] data,next_data;
reg rx_done;

// Actualización de variables
always @(posedge i_clk,posedge i_reset) begin

    if (i_reset) begin
        rx_state <= RX_IDLE_STATE;
        data_counter <= 0;
        ticks_counter <= 0;
        data <= {NB_DATA{1'b0}};
    end

    else begin
        rx_state <= next_rx_state;
        data_counter <= next_data_counter;
        ticks_counter <= next_ticks_counter;
        data <= next_data;
    end

end
```

El bloque combinacional `always @(*)`, por otra parte implementa la lógica del siguiente estado y el control de salida. Al inicio de este bloque, todas las variables del siguiente estado se inicializan con sus valores actuales y `rx_done` se reinicia a cero para habilitar una nueva recepción.

```
//Lógica del siguiente estado
always @(*)begin

    next_rx_state = rx_state;
    next_data_counter = data_counter;
    next_ticks_counter = ticks_counter;
    next_data = data;
    rx_done = 1'b0;

    case(rx_state)
```

En el estado `RX_IDLE`, el receptor espera a que la línea pase a nivel bajo (0), lo que indica la llegada de un bit de start. Cuando esto ocurre (`i_rx` = 0), asume que ha comenzado una

nueva recepción, pone el contador de ticks a 0 y pasa al estado `RX_START_STATE` para verificar el bit de inicio.

```
RX_IDLE_STATE:begin

    if(~i_rx) begin
        next_rx_state = RX_START_STATE;
        next_ticks_counter = 4'b0;
    end

end
```

En `RX_START_STATE`, se cuentan los ticks hasta alcanzar el punto medio del bit de inicio. Dado que el bit de inicio dura 16 ticks, el receptor espera hasta que el contador de pulsos alcance 7, lo que significa que se encuentra aproximadamente a la mitad del tiempo del bit. A continuación, reinicia los contadores de ticks y de datos, pone el registro de datos a 0 y pasa al estado `RX_DATA_STATE` para comenzar a muestrear los bits de datos.

```
RX_START_STATE:begin

    if(i_s_tick) begin

        if(ticks_counter == 7)begin
            next_rx_state = RX_DATA_STATE;
            next_ticks_counter = 4'b0;
            next_data_counter = 3'b0;
            next_data={NB_DATA{1'b0}};

        end

        else begin
            next_ticks_counter = ticks_counter+1;
        end
    end

end
```

En el estado `RX_DATA_STATE`, el receptor muestrea cada bit de datos. Con cada `i_s_tick`, el contador de ticks se incrementa y, al llegar a 15,el receptor muestrea el valor actual de `i_rx`, que representa el siguiente bit de datos entrante. El bit muestreado se copia al registro de datos mediante la operación `{i_rx, data[NB_DATA-1:1]}`, colocando así el nuevo bit en la posición más significativa y desplazando los bits anteriores hacia la derecha. A continuación, se incrementa el contador de datos para llevar la cuenta de los bits recibidos. Cuando se han recibido todos los bits (es decir, `data_counter == NB_DATA-1`), el receptor pasa al estado `RX_STOP_STATE` para verificar el bit desktop.

```
RX_DATA_STATE:begin

    if(i_s_tick) begin

        if(ticks_counter == 15)begin

            next_ticks_counter=4'b0;
            next_data={i_rx,data[NB_DATA-1:1]};

            if(data_counter == NB_DATA-1) begin
                next_rx_state = RX_STOP_STATE;
                next_data_counter = 3'b0;
            end

            else begin
                next_data_counter = data_counter + 1;
            end

        end

        else begin
             next_ticks_counter = ticks_counter+1;
        end

    end

end
```

El estado `RX_STOP_STATE` se encarga de validar el bit de stop y completar la recepción de la trama. Este bit de parada debe estar en nivel lógico alto (1) durante al menos un período de bit. Cuando el contador de ticks alcanza `STOP_TICKS - 1`, lo que significa que ha transcurrido una duración completa del bit de stop, se reinicia el contador de ticks, el receptor regresa al estado `RX_IDLE_STATE` y `rx_done` se pone en alto para indicar que se ha recibido una trama de datos válida.

```
RX_STOP_STATE:begin

    if(i_s_tick) begin

        if (ticks_counter == (STOP_TICKS-1))begin
            next_ticks_counter = 4'b0;
            next_rx_state = RX_IDLE_STATE;
            rx_done = 1'b1;
        end

        else begin
            next_ticks_counter = ticks_counter + 1;
        end

    end

end
```

Al final del módulo, se asignan las salidas. La salida `o_rx_data` proporciona el byte de datos almacenado en el registro de datos, mientras que `o_rx_done_tick` se asocia a `rx_done`.

```
assign o_rx_data = data;
assign o_rx_done_tick = rx_done;
```

tx_mod

Este módulo Verilog implementa el transmisor UART. Al igual que el transmisor, el diseño se basa en una máquina de estados finitos (FSM) que controla las diferentes etapas de transmisión. La tarea del módulo es tomar un byte de datos y enviarlo en serie, bit a bit, de acuerdo al mismo formato de trama que se mostraba para el receptor

Al igual que en el receptor se definen el número de bits de datos y de stop con los parámetros `NB_DATA` y `STOP_TICKS` respectivamente. Las entradas del módulo incluyen el reloj principal `i_clk`, el clock del generador de baudios `i_s_tick` que define la

velocidad de transmisión, una señal `i_tx_start` para iniciar la transmisión, el byte de datos a transmitir `i_tx_data` y una entrada de reseteo `i_reset`. Las salidas son `o_tx_done_tick`, la cual que indica que la transmisión se ha completado, y `o_tx`, que es la línea por donde se transmite cada bit de datos en serie.

```
1   module tx_mod
2   #(
3
4       parameter NB_DATA = 8,
5       parameter STOP_TICKS = 16
6
7   )
8   (
9
10      input wire i_clk,
11      input wire i_s_tick,
12      input wire i_tx_start,
13      input wire [NB_DATA-1:0] i_tx_data,
14      input wire i_reset,
15
16      output wire o_tx_done_tick,
17      output wire o_tx
18
```

La primera sección define la codificación de los cuatro estados de la FSM:

- TX_IDLE_STATE,
- TX_START_STATE,
- TX_TRANSMIT_STATE
- TX_STOP_STATE.

El transmisor comienza en estado inactivo, esperando uan señal de inicio. Una vez transcurrido el bit de start, se procede a enviar cada bit de datos en serie. Finalmente, se envían uno o más bits de stop antes de volver al estado inactivo. Estos estados garantizan una secuencia de eventos precisa y un correcto funcionamiento del transmisor.

```
//Estados
localparam TX_IDLE_STATE= 2'b00;
localparam TX_START_STATE= 2'b01;
localparam TX_TRANSMIT_STATE=2'b10;
localparam TX_STOP_STATE=2'b11;
```

A continuación, se declaran varios registros para monitorear el estado de funcionamiento del transmisor.

- `tx_state` y `next_tx_state` almacenan los estados actual y siguiente de la máquina de estados finitos (FSM).

- `data_counter` y `next_data_counter` cuentan los bits de datos enviados hasta el momento, asegurando que se transmitan exactamente `NB_DATA` bits.
- `ticks_counter` y `next_ticks_counter` llevan el recuento de la cantidad de ticks del generador de baudios.
- Los registros `` `data` `` y `` `next_data` `` almacenan los datos que se van a transmitir y los desplazan a la derecha después de enviar cada bit.
- `tx_reg` y `tx_next` almacenan el bit de salida actual y el siguiente valor que se enviará a la línea TX, mientras que `tx_done` indica el final de la transmisión.

```
// Registros de estado,contador de data,contador de ticks y data
reg[1:0] tx_state,next_tx_state;
reg[2:0] data_counter,next_data_counter;
reg[3:0] ticks_counter,next_ticks_counter;
reg[NB_DATA-1:0] data,next_data;
reg tx_done;
reg tx_reg , tx_next;
```

El bloque secuencial, sensible a los flancos ascendentes de `i_clk` e `i_reset`, simplemente actualiza todos estos registros. Si se produce un reinicio, el transmisor vuelve al estado de reposo con la línea de salida (`tx_reg`) en nivel lógico alto

```
// Actualización de variables
always @(posedge i_clk,posedge i_reset) begin

    if (i_reset) begin
        tx_state <= TX_IDLE_STATE;
        data_counter <= 0;
        ticks_counter <= 0;
        data <= {NB_DATA{1'b0}};
        tx_reg <= 1'b1;
    end

    else begin
        tx_state <= next_tx_state;
        data_counter <= next_data_counter;
        ticks_counter <= next_ticks_counter;
        data <= next_data;
        tx_reg <= tx_next;
    end

end
```

El bloque combinacional `always @(*)` define la lógica del siguiente estado para la máquina de estados finitos (FSM). Primero, se actualizan todas las señales "next" con sus valores actuales y establece `tx_done` en cero para permitir una nueva transmisión. A continuación, evalúa las transiciones de estado y los valores de salida de la FSM.

En el estado `TX_IDLE`, el transmisor mantiene la salida del transmisor en alto (1) y chequea el valor de `i_tx_start`. Si se encuentra en 1, se cargan los bits de entrada(`i_tx_data`) en el registro de datos, se reinicia el contador de ticks y pasa al estado `TX_START_STATE`.

```
TX_IDLE_STATE:begin

    tx_next = 1'b1;
    if(i_tx_start)begin
        next_tx_state = TX_START_STATE;
        next_ticks_counter = 4'b0;
        next_data = i_tx_data;
    end
end
```

En el estado `TX_START_STATE`, la línea TX se pone a 0 para generar el bit de inicio. El sistema cuenta los pulsos `i_s_tick` para medir la duración de un bit. Una vez transcurridos 16 ticks (un periodo de bit completo), se reinicia el contador de ticks y el contador de datos, y pasa al estado `TX_TRANSMIT_STATE` para comenzar a enviar bits de datos.

```
TX_START_STATE:begin

    tx_next = 1'b0;
    if(i_s_tick)begin

        if(ticks_counter == 15)begin
            next_ticks_counter = 4'b0;
            next_data_counter = 3'b0;
            next_tx_state = TX_TRANSMIT_STATE;
        end

        else begin
            next_ticks_counter = ticks_counter +1;
        end

    end

end
```

En el estado `TX_TRANSMIT`, el bit menos significativo del registro de datos (`data[0]`) se coloca en `tx_next` para transmitirse. En cada iteración, el contador de ticks se incrementa hasta llegar a 15, lo que marca el final del bit actual. Cuando esto ocurre, los datos se

desplazan una posición a la derecha para colocar el siguiente bit a transmitir en `data[0]` y el contador de datos se incrementa. Si se han transmitido todos los bits de datos (cuando `data_counter == NB_DATA - 1`), la máquina de estados finitos pasa al estado `TX_STOP_STATE`.

```
TX_TRANSMIT_STATE:begin

    tx_next = data[0];

    if(i_s_tick)begin

        if(ticks_counter == 15)begin
            next_ticks_counter = 4'b0;
            next_data = data >> 1;
            if(data_counter == (NB_DATA-1))begin
                next_data_counter = 3'b0;
                next_tx_state = TX_STOP_STATE;
            end

            else begin
                next_data_counter = data_counter + 1;
            end

        end

        else begin
            next_ticks_counter = ticks_counter + 1;
        end

    end

end
```

Finalmente, en el estado `TX_STOP_STATE`, la salida vuelve a 1, lo que representa el bit de stop. Se cuenta `STOP_TICKS` para mantener la duración correcta del bit. Cuando finaliza este periodo, indica que la transmisión se ha completado. Por lo tanto,se establece `tx_done` en 1 y se vuelve al estado `TX_IDLE_STATE` para esperar la siguiente solicitud de transmisión.

```
TX_STOP_STATE:begin

    tx_next = 1'b1;

    if (i_s_tick)begin

        if(ticks_counter == (STOP_TICKS-1))begin
            next_tx_state = TX_IDLE_STATE;
            tx_done= 1'b1;
        end

        else begin
            next_ticks_counter = ticks_counter + 1;
        end

    end

end
```

Las asignaciones finales conectan las señales internas con las salidas del módulo. La salida `o_tx` es controlada por `tx_reg`, mientras que `o_tx_done_tick` se asigna a `tx_done`, que indica la finalización de la transmisión.

```
assign o_tx = tx_reg;
assign o_tx_done_tick = tx_done;
```

interface_mod.v

Este módulo Verilog define una interfaz que coordina la comunicación entre el receptor, el transmisor y la ALU. Su función es interpretar los datos recibidos a través de los módulos UART, actualizar los registros internos o las señales de control de la ALU y, cuando se solicita, transmitir el resultado de la ALU de vuelta a través del transmisor UART.

El módulo se parametriza mediante 2 constantes:

- `NB_DATA`: define el ancho (en bits) de los datos transmitidos y recibidos
- `NB_ALU_OP`: especifica cuántos bits se utilizan para representar los códigos de operación de la ALU

Las señales de entrada se dividen en tres grupos funcionales. El primer grupo corresponde al receptor UART

- `i_rx_data` transporta los datos recibidos del módulo RX UART, mientras que
- `i_rx_done` indica que se ha recibido un byte completo y está listo para su interpretación.

El segundo grupo corresponde a la ALU:

- `i_alu_res` transporta el resultado de una operación de la ALU que puede enviarse de vuelta a través del transmisor UART.

Finalmente, `i_clk` e `i_reset` son las señales de reloj y reinicio que controlan el funcionamiento del módulo.

Las salidas se agrupan de forma similar:

- `o_tx_start` y `o_tx_data` son las señales de control y datos respectivamente enviadas al transmisor UART, iniciando una transmisión y proporcionando los datos que se enviarán.

- `o_alu_OP`, `o_alu_A` y `o_alu_B` son las señales enviadas a la ALU, que representan el código de operación actual y los operandos A y B.

El funcionamiento interno de este módulo se organiza como una FSM de tres estados:

- `INTF_IDLE_STATE`,
- `INTF_SEND_STATE`
- `INTF_RECEIVE_STATE`.

La interfaz se inicia en el estado inactivo, a la espera de nuevos comandos del receptor UART. Según el comando recibido, se prepara para recibir datos adicionales (para configurar operandos o códigos de operación) o envía datos de vuelta a través del transmisor UART (para transmitir el resultado de la ALU).

```
module interface
#(

    parameter NB_DATA    = 8,
    parameter NB_ALU_OP  = 6,
    parameter NB_INTF_OP = 8
)
(
    input wire                  i_clk,
    input wire                  i_reset,
    input wire  [NB_DATA-1:0]   i_rx_data,
    input wire                  i_rx_done,

    input wire  [NB_DATA-1:0]   i_alu_res,

    output wire                 o_tx_start,
    output wire [NB_DATA-1:0]   o_tx_data,

    output wire [NB_ALU_OP-1:0] o_alu_OP,
    output wire [NB_DATA-1:0]   o_alu_A,
    output wire [NB_DATA-1:0]   o_alu_B
);

// 3 possible states:
// 00 - Idle
// 01 - Send
// 10 - Receive

localparam INTF_IDLE_STATE    = 2'b00;
localparam INTF_SEND_STATE    = 2'b01;
localparam INTF_RECEIVE_STATE = 2'b10;
```

Se definen cuatro códigos de operación para especificar los distintos comandos que la interfaz puede interpretar.

- `ALU_OP_SET_A` (b00000001): indica a la interfaz que reciba un nuevo valor para el operando A.
- `ALU_OP_SET_B` (b00000010): indica que se debe actualizar el operando B.
- `ALU_OP_SET_OP` (b00000011): indica a la interfaz que el siguiente byte recibido representa un código de operación de la ALU, que define qué operación aritmética o lógica debe realizar .
- `ALU_OP_GET_RES` (b00000000): indica a la interfaz que envíe el resultado de la ALU a través del transmisor UART.

Varios registros internos almacenan tanto el estado como los datos:

.

- `intf_state` y `next_intf_state` representan los estados actual y siguiente de la FSM
- `intf_opcode` y `next_intf_opcode` almacenan el codigo de operación actual y siguiente de la interfaz
- `alu_data_A`, `alu_data_B` y `alu_opcode` almacenan los operandos actuales y el código de operación que se enviarán a la ALU.
- `tx_start` y `tx_data` se utilizan para controlar y proporcionar datos al transmisor UART. Cada registro tiene una versión "next" correspondiente, utilizada en la lógica

combinacional para calcular los nuevos valores que se cargarán en el siguiente ciclo de reloj.

```
// Operations 1-3 change state from IDLE to SEND
// Operation 4 change state from IDLE to RECEIVE

localparam ALU_OP_GET_RES = 8'b00000000;
localparam ALU_OP_SET_A   = 8'b00000001;
localparam ALU_OP_SET_B   = 8'b00000010;
localparam ALU_OP_SET_OP  = 8'b00000011;

reg [1:0]          intf_state,  next_intf_state;
reg [NB_DATA-1:0]  intf_opcode, next_intf_opcode;

reg [NB_DATA-1:0]  alu_data_A,  next_alu_data_A,
                   alu_data_B,  next_alu_data_B;
reg [NB_ALU_OP-1:0] alu_opcode, next_alu_opcode;

reg                tx_start,    next_tx_start;
reg [NB_DATA-1:0]  tx_data,     next_tx_data;
```

El bloque `always` secuencial, activado por el flanco ascendente del clock, se encarga de actualizar todos los registros internos. En caso de reset, todos los registros se limpian y se configura el estado a `INTF_IDLE_STATE`. En condiciones normales de funcionamiento, el bloque asigna a cada registro su valor “next”correspondiente.

```
always @(posedge i_clk) begin

    if(i_reset) begin
        intf_state   <= INTF_IDLE_STATE;
        intf_opcode  <= 8'b0;

        alu_data_A   <= {NB_DATA{1'b0}};
        alu_data_B   <= {NB_DATA{1'b0}};
        alu_opcode   <= {NB_ALU_OP{1'b0}};

        tx_start     <= 1'b0;
        tx_data      <= {NB_DATA{1'b0}};
    end

    else begin

        intf_state   <= next_intf_state;
        intf_opcode  <= next_intf_opcode;

        alu_data_A   <= next_alu_data_A;
        alu_data_B   <= next_alu_data_B;
        alu_opcode   <= next_alu_opcode;

        tx_start     <= next_tx_start;
        tx_data      <= next_tx_data;
    end
```

El bloque `always` combinacional define la lógica principal de la máquina de estados, determinando cómo reacciona la interfaz a las señales de entrada y las transiciones entre estados. Al inicio de este bloque, todos los valores de "next" se inicializan con sus valores actuales, y la señal `next_tx_start` se pone a cero para evitar transmisiones no deseadas. A continuación, una instrucción `case` gestiona el comportamiento de cada estado de la FSM.

```
always @(*) begin

    next_intf_state  = intf_state;
    next_intf_opcode = intf_opcode;
    next_alu_data_A  = alu_data_A;
    next_alu_data_B  = alu_data_B;
    next_alu_opcode  = alu_opcode;
    next_tx_data     = tx_data;
    next_tx_start    = 1'b0;
```

En el estado `INTF_IDLE_STATE`, la interfaz espera la llegada de un nuevo byte del receptor UART, indicado por `i_rx_done`. Cuando esto sucede, el byte recibido `i_rx_data` se interpreta como un comando o código de operación y se almacena en `next_intf_opcode`. El valor de este código de operación determina el siguiente estado. Si el código de operación corresponde a `ALU_OP_SET_A`, `ALU_OP_SET_B` o `ALU_OP_SET_OP`, la interfaz pasa al estado `INTF_RECEIVE_STATE`, ya que pronto recibirá otro byte con el operando o el código de operación. Si el código de operación corresponde a `ALU_OP_GET_RES`, la interfaz pasa al estado `INTF_SEND_STATE` para transmitir el resultado de la ALU a través del transmisor UART. Cualquier código de operación no reconocido deja la máquina de estados finitos en estado inactivo.

```
INTF_IDLE_STATE: begin

    if(i_rx_done) begin
        next_intf_opcode = i_rx_data;

        case(i_rx_data)
            ALU_OP_SET_A: begin
                next_intf_state = INTF_RECEIVE_STATE;
            end

            ALU_OP_SET_B: begin
                next_intf_state = INTF_RECEIVE_STATE;
            end

            ALU_OP_SET_OP: begin
                next_intf_state = INTF_RECEIVE_STATE;
            end

            ALU_OP_GET_RES: begin
                next_intf_state = INTF_SEND_STATE;
            end

            default: begin
                next_intf_state = INTF_IDLE_STATE;
            end
        endcase
    end
end
```

En el estado `INTF_SEND_STATE`, la interfaz se prepara para transmitir el resultado de la ALU. Carga el resultado de la ALU (i_alu_res) en next_tx_data y activa next_tx_start, indicando al transmisor UART que comience a enviar datos. Tras iniciar esta transmisión, la máquina de estados finitos (FSM) vuelve inmediatamente al estado IDLE, lista para recibir nuevos comandos.

```
INTF_SEND_STATE: begin

    next_tx_data = i_alu_res;
    next_tx_start = 1'b1;
    next_intf_state = INTF_IDLE_STATE;

end
```

En el estado `INTF_RECEIVE_STATE` la interfaz espera la llegada del siguiente byte, que contendrá los datos asociados al comando recibido previamente. Cuando `i_rx_done` se activa, la interfaz lee `i_rx_data` y, según el código de operación almacenado (`intf_opcode`), actualiza el registro correspondiente. Si el último comando fue `ALU_OP_SET_A`, el byte recibido se almacena en alu_data_A. Si fue `ALU_OP_SET_B`, el byte se almacena en alu_data_B. Si se tratara de `ALU_OP_SET_OP`, el byte se almacena en `alu_opcode`, pero solo se conservan los bits correspondientes al tamano definido por `NB_ALU_OP`, lo que garantiza la compatibilidad con el ancho del código de operación de la ALU. Tras procesar estos datos entrantes, la máquina de estados finitos vuelve al estado `INTF_IDLE_STATE`, completando el ciclo de recepción.

```
INTF_RECEIVE_STATE: begin

    if(i_rx_done) begin

        case(intf_opcode)

            ALU_OP_SET_A: begin
                next_alu_data_A = i_rx_data;
            end

            ALU_OP_SET_B: begin
                next_alu_data_B = i_rx_data;
            end

            ALU_OP_SET_OP: begin
                next_alu_opcode = i_rx_data[NB_ALU_OP-1:0];
            end

            default: begin
                // Do nothing
            end

        endcase

        next_intf_state = INTF_IDLE_STATE;
    end

end
```

Finalmente, se asignan las salidas del módulo.

```
assign o_tx_start  = tx_start;
assign o_tx_data   = tx_data;

assign o_alu_OP    = alu_opcode;
assign o_alu_A     = alu_data_A;
assign o_alu_B     = alu_data_B;
```

top.v

Este bloque se encarga de instanciar todos los submódulos anteriores para permitir la operación del sistema que se mostraba en el diagrama de la descripción . Comienza definiendo los parámetros `NB_DATA` (ancho de datos) y `NB_ALU_OP` (ancho del código de operación de la ALU). Luego , define las señales internas que van a ser compartidas por los destinos módulos: `tick`, `rx_data`, `tx_data`, `rx_done`, `tx_start` y `tx_done`.

```
module top
#(
    parameter NB_DATA   = 8,
    parameter NB_ALU_OP = 6
)
(
    input wire   i_clk,
    input wire   i_reset,
    input wire   i_rx,
    output wire  o_tx
);

wire tick;
wire [NB_DATA-1:0] rx_data;
wire [NB_DATA-1:0] tx_data;
wire rx_done;
wire tx_start;
wire tx_done;

wire [NB_ALU_OP-1:0] alu_opcode;
wire [NB_DATA-1:0] alu_data_A;
wire [NB_DATA-1:0] alu_data_B;
wire [NB_DATA-1:0] alu_result;
```

Una vez hecho esto, procede a inicializar cada módulo uno por uno.

tp2_constrainsts.xdc

Configura los pines del transmisor y el receptor UART, el clock y el reset.

```
#Clock
set_property -dict { PACKAGE_PIN W5 IOSTANDARD LVCMOS33 } [get_ports { i_clk }];

# Reset
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { i_reset }];

## USB-RS232
set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports i_rx]
set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVCMOS33 } [get_ports o_tx]
```

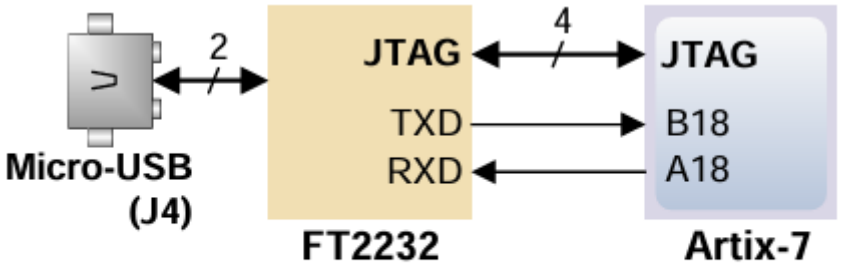
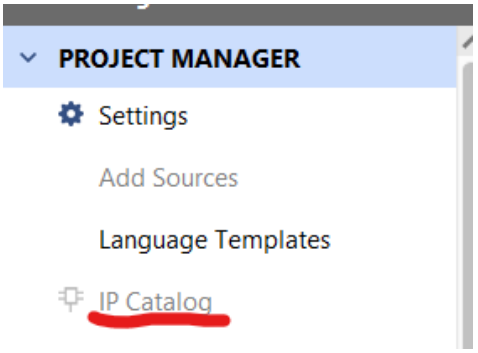


Figure 6. Basys 3 FT2232HQ connections.

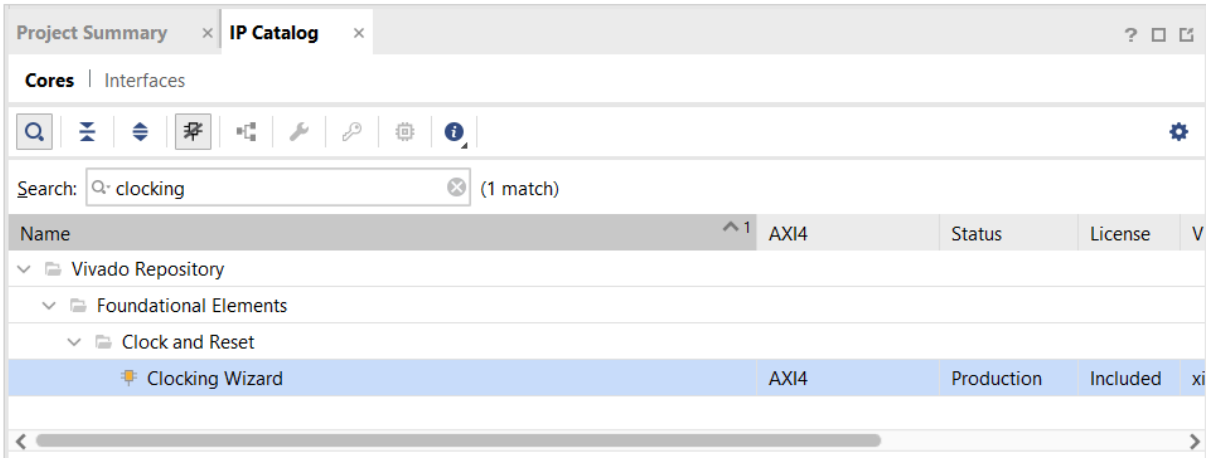
clk_wz_0.veo

Este archivo, tiene como función manipular el valor de clock del sistema por defecto (100MHz) y reducirlo a la mitad . A continuación, se detalla el paso a paso para generar el archivo en Vivado

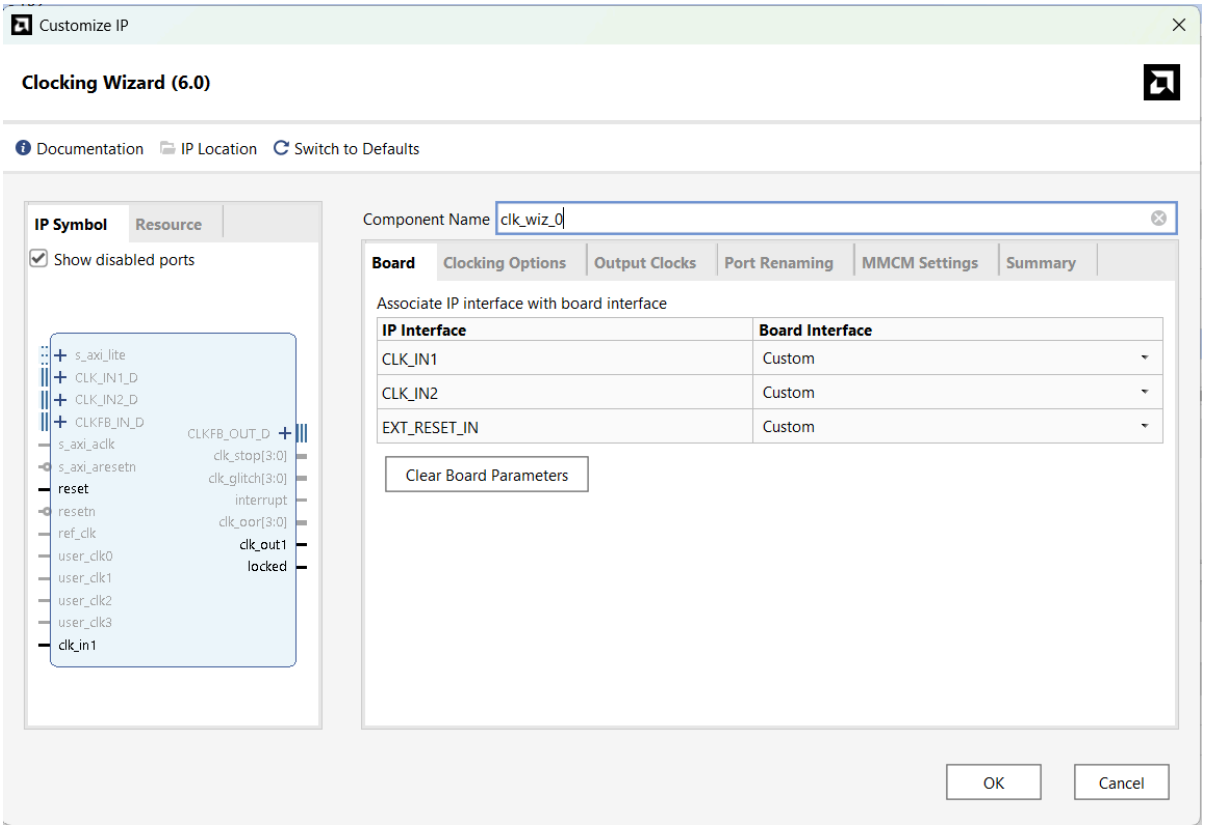
1) Dirigirse a la sección **IP Catalog** debajo de **PROJECT MANAGER**

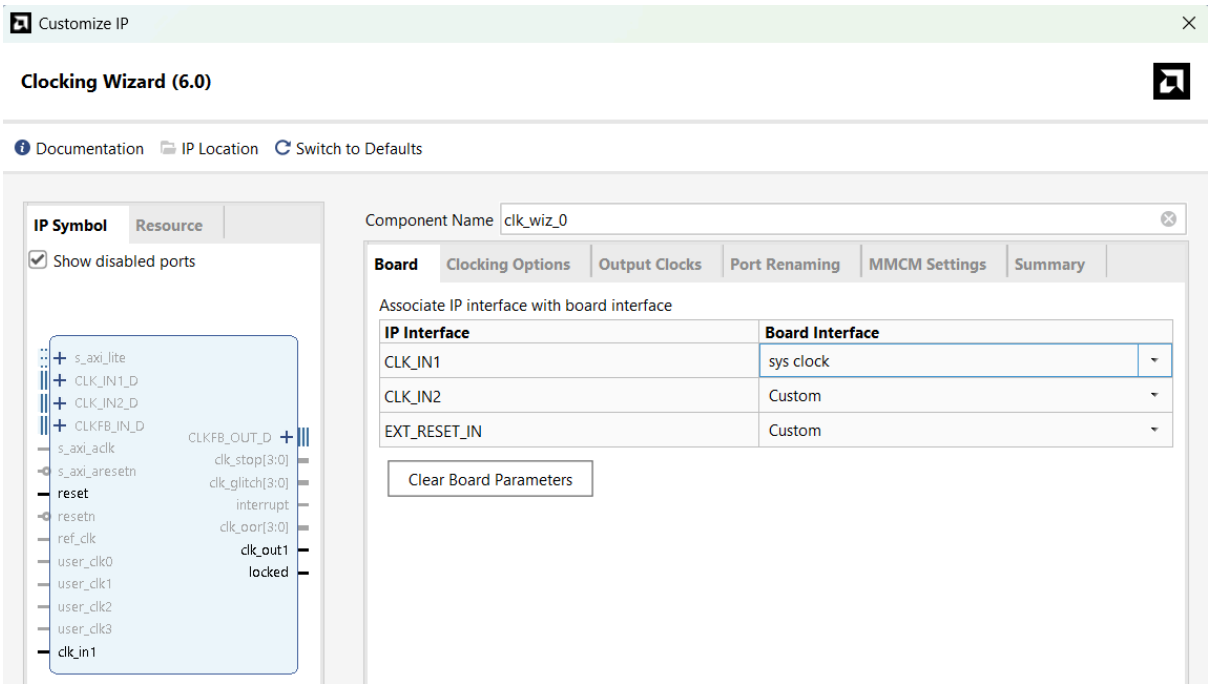


2) Buscar “clocking”en la barra de búsqueda,desplegar la sección **Clock and Reset** y clickear sobre **Clocking Wizard**

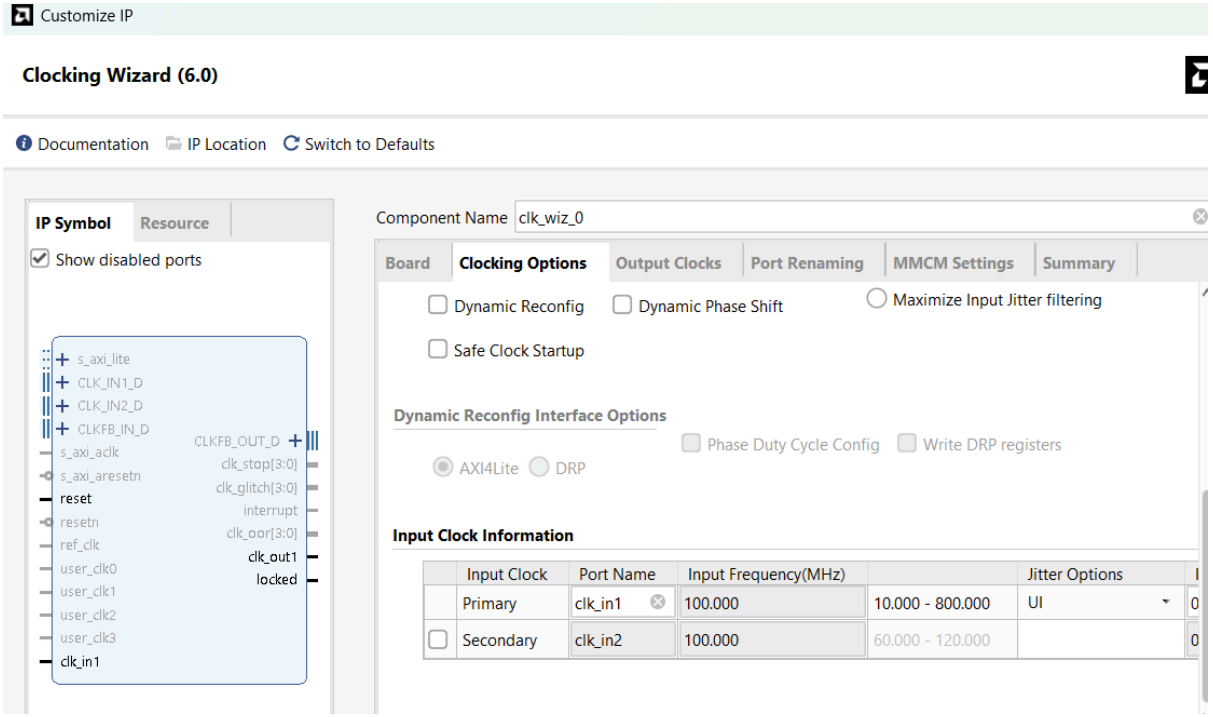


3)En el cuadro que se despliega, modificar el campo **CLK_IN1** bajo la columna de **IP interfaces** de “custom” a “sys_clock”

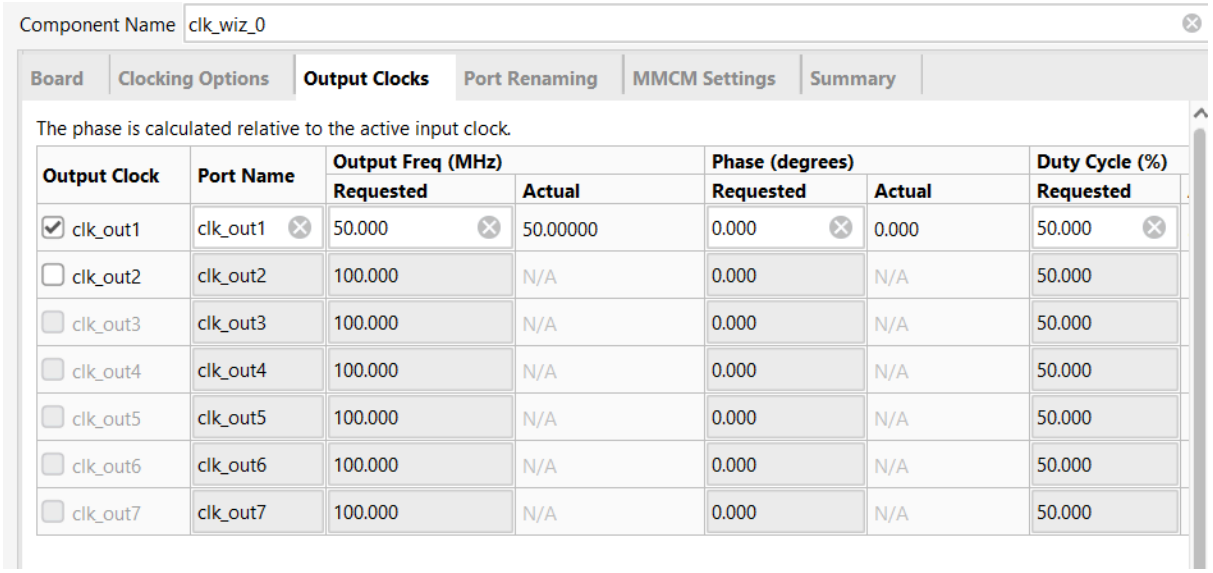




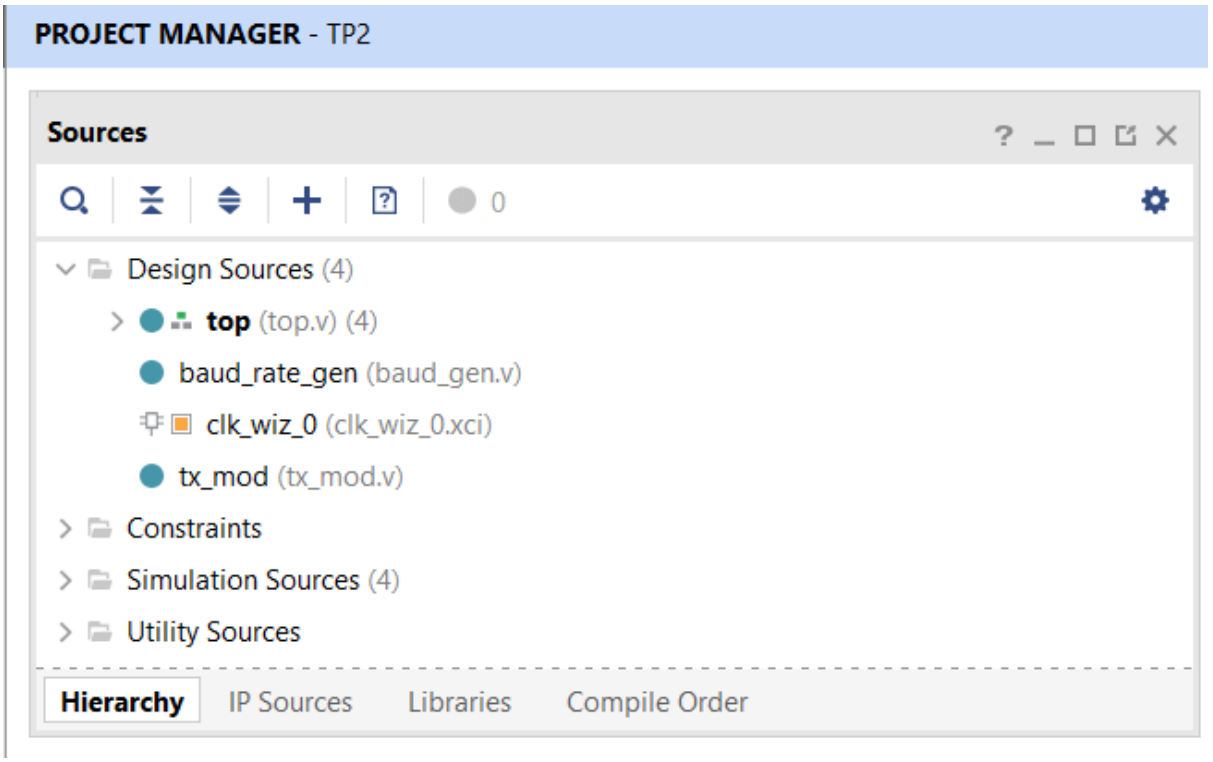
4)Clickear sobre la pestaña **clocking options** y chequear que el primary input clock esté configurado a una frecuencia de 100MHz.



5)Clickear en la pestaña output clocks y modificar **clk_out1** a 50MHz y guardar los cambios con **Ok.**



Una vez hecho lo anterior,se genera el archivo en el proyecto:



Al clickear sobre el mismo se observa la implementación