UNL • FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

ELECTRÓNICA DIGITAL

2023

Alumnos:

- Bargas Darío Santiago
- Mariño Pablo

Tema: Implementación y programación de algoritmo divisor en una FPGA.

Resumen:

El presente informe integrador final tiene como objetivo mostrar y dar explicación del diseño e implementación de un divisor de cuatro bits de números enteros. Para ello utilizamos una FPGA (dispositivo semiconductor programable que se utiliza para implementar circuitos digitales personalizados), donde con el lenguaje de descripción de hardware llamado Verilog, la programamos. Se seguirá un algoritmo especifico y se utilizaran los recursos de la FPGA para la prueba del mismo.

Palabras claves: FPGA, FSM, Contador, divisor, LEDs, bits, top.

1. Introducción:

La consigna de este Trabajo Integrador Final consiste en realizar un divisor de números enteros de cuatro bits mediante el uso de una FSM (Finite State Machine), compuesta por:

- Módulo de procesamiento de datos que permita al usuario cargar los valores del numerador, denominador y obtener el resultado.
- Bloques combinacionales y/o secuenciales que procesan la información.
- FSM que realice la operación de división entera.

La cátedra propuso el siguiente algoritmo:

1. Cargar los datos, en este caso numerador y denominador de la división.
2. Determinar si el denominador es distinto de cero. Si el denominador fuera igual a cero, la división no sería posible. En este caso el proceso termina, pudiendo informar esta situación mediante algún mensaje o señal de error. De lo contrario, es posible realizar la división.
3. Para obtener el resultado y el resto de la división, es posible restar al numerador una cantidad igual al denominador, iterando sucesivamente hasta que el numerador sea menor al denominador. Cuando esto sucede, el resultado es igual al número de iteraciones realizadas, y el resto de la división es igual al valor del numerador tras la última iteración.

En este trabajo integrador final, se propone implementar un divisor de números enteros de cuatro bits utilizando una FPGA y una FSM. La FSM estará compuesta por un módulo de procesamiento de datos que permitirá al usuario cargar los valores del numerador y denominador, y obtener el resultado de la división. La operación de división entera se realizará siguiendo un algoritmo específico.

El algoritmo propuesto consiste en cargar los datos del numerador y denominador, verificar si el denominador es distinto de cero (si no lo es, la división no es posible y se mostrará un mensaje de error), y luego iterar sucesivamente restando al numerador el valor del denominador hasta que el numerador sea menor al denominador. El resultado de la división será igual al número de iteraciones realizadas, y el resto de la división será igual al valor del numerador después de la última iteración.

Para implementar este algoritmo, se utilizará una FSM que representará el comportamiento del sistema. Se diseñará un diagrama de estados y transiciones que describe el flujo de la FSM. Además, se determinarán los elementos de hardware necesarios para llevar a cabo el algoritmo propuesto.

El diseño se implementará en Verilog, un lenguaje de programación de hardware, y se comprobará su funcionamiento mediante testbench. Finalmente, se impactará el diseño en la FPGA proporcionada por la cátedra, utilizando los LEDs para visualizar el numerador, denominador y resultado, y los botones para realizar las operaciones de carga y visualización de resultados.

2. Desarrollo:

Antes de dar la explicación de diagramas y figuras realizadas, daremos una breve explicación y un ejemplo del algoritmo de división (Figura 1) que usamos y programamos. Imaginemos que queremos dividir 7 dividido 2:

- Primero Inicializamos los datos: Numerador=7 Denominador=2 Resto=7 Cociente=0 donde numerador y denominador son los datos ingresados, el resto es igual al numerador y el cociente es 0 que son los datos inicializados automáticamente cuando comienza a funcionar el algoritmo.
- Luego chequeamos que el Numerador sea mayor al Denominador para seguir iterando, si no cumple esa condición se corta la división. Si se cumple la condición, el Numerador ahora será igual al numerador anterior menos el denominador. El resto será igual al resto anterior menos el denominador. El cociente será igual al cociente anterior más uno y el denominador siempre se mantiene igual.
- Si el Numerador < Denominador, cortamos la división y mostramos el cociente y el resto. Se vio en el primer caso que al resto se le asigno el valor del numerador, y esto es porque si en la primera iteración se cumple la condición mencionada, el resto ya tiene su valor y el cociente también, que serian 0 y 7 respectivamente.

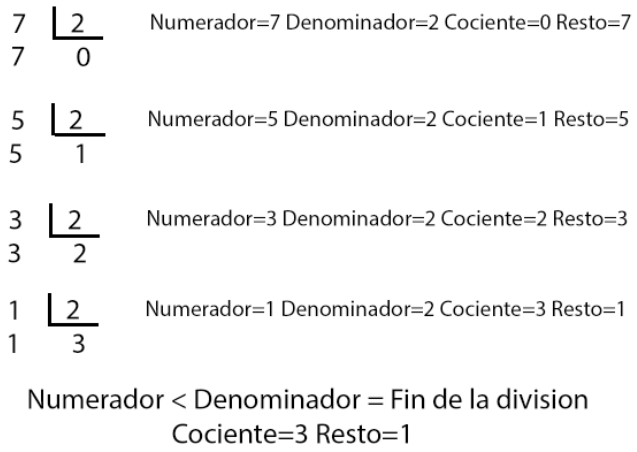


Figura 1. Ejemplo del algoritmo para la división

Una vez comprendido el algoritmo de división y visto el respectivo ejemplo anteriormente, procederemos a mostrar el diagrama de flujo (Figura 2). El mismo nos mostrara el funcionamiento de nuestra FSM.

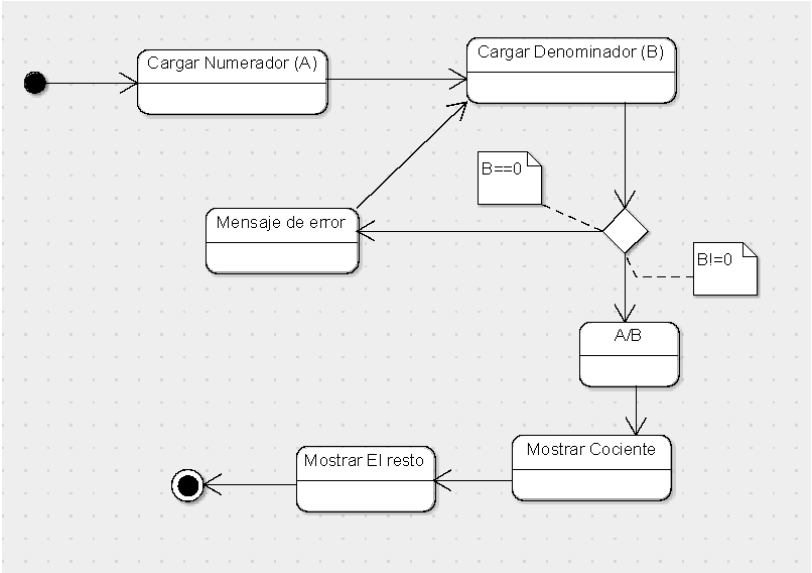


Figura 2. Diagrama de flujo de datos

A continuación, se describe cada uno de los pasos:

- Paso 1: Cargar numerador
- Paso 2: Cargar denominador
- Paso 3: Si el denominador es igual a cero, mostramos un mensaje de error y vamos a cargar el denominador nuevamente
- Paso 4: Si el denominador es distinto de cero vamos a realizar la división de A/B
- Paso 5: Mostramos el cociente
- Paso 6 Mostramos el resto
- Paso 7: Finaliza la división

Una vez planteado el proceso en un diagrama de flujo, se establecieron las decisiones de diseño para lograr una implementación eficiente y funcional.

El divisor comenzará en la carga de datos del numerador. Se podrá ingresar un valor del 0 al 15 en binario utilizando los botones de la FPGA. El número correspondiente seleccionado se irá mostrando en la FPGA con los LEDs encendidos, una vez que el contador llega al número 15 vuelve al 0. Cuando se confirma el valor requerido, el divisor pasará a la acción de elegir un valor para el numerador, con la misma lógica mencionada anteriormente. Si se selecciona un valor igual a 0 en el denominador se mostrará un mensaje de error, este mensaje será mostrado en el cociente y resto como el número 15 en la FPGA, es decir todos los LEDs encendidos. Se podrá apretar el botón OK para volver al estado de cargar numerador o también se podrá optar por apretar el botón de resetear para volver al estado de cargar numerador.

Una vez validado el valor del denominador, el sistema realizará las operaciones asignadas y pasará a la muestra de resultados. En la misma se mostrará primero el cociente de la división a través de los LEDs de la FPGA. Si se pasa al siguiente estado el sistema mostrará el resto de la división, una vez mostrado el resto, se puede volver al estado de cargar otros valores para el numerador y denominador.

Como la FPGA consta de 4 botones, el control de los mismos se definió de la siguiente manera:

- **BOTÓN 1:** Incremento de los valores a ingresar.
- **BOTÓN 2:** Decremento de los valores a ingresar.
- **BOTÓN 3:** Ok, avanza al siguiente paso.
- **BOTON 4:** Reiniciar.

Para la implementación de la FSM, se pensaron diferentes módulos que ayudaron a completar lo que seria el divisor completo. Hay varios caminos y opciones a la hora de realizar estos algoritmos y testear su funcionalidad en test-bench, sin embargo, a la hora de impactarlo en una FPGA pueden ocurrir bugs o ambigüedades y podría influir en su funcionamiento. Para evitar estos problemas se respetó la sintaxis del libro “FPGA Prototyping By Verilog Examples”, garantiza consistencia y estabilidad en el circuito impactado.

A continuación, se presentarán los mismos con su respectivo gráfico:

- **Módulo contador:**

Se comenzó programando el módulo contador, donde su funcionalidad es incrementar y decrementar valores entre 0 y 15 a través de una FSM.

Este módulo tiene una entrada de reloj (**clk**), una entrada de reinicio (**reset**), una entrada ascendente (**up**), una entrada descendente (**down**), y una salida de contador de 4 bits (**count**).

El módulo contiene un registro de 4 bits llamado **count_reg** que almacena el valor actual del contador. El contador comienza en cero (**count_reg = 0**) al inicio.

La lógica principal del módulo se encuentra dentro del bloque **always @(posedge clk)**, que se activa en el flanco de subida del reloj. Aquí es donde ocurre el incremento y decremento del contador.

- Si la señal **reset** está activa, el contador se reinicia y se establece en cero (**count_reg <= 0**).
- Si la señal **up** está activa, el contador se incrementa en uno (**count_reg <= count_reg + 1**).
- Si la señal **down** está activa, el contador se decrementa en uno (**count_reg <= count_reg - 1**).

Finalmente, la asignación **assign count = count_reg** conecta la salida **count** con el valor actual del contador **count_reg**.

En resumen, este código representa un contador de 4 bits que puede incrementarse o decrementarse utilizando las señales **up** y **down**, respectivamente. Además, se proporciona una señal de reinicio **reset** para restablecer el contador a cero.

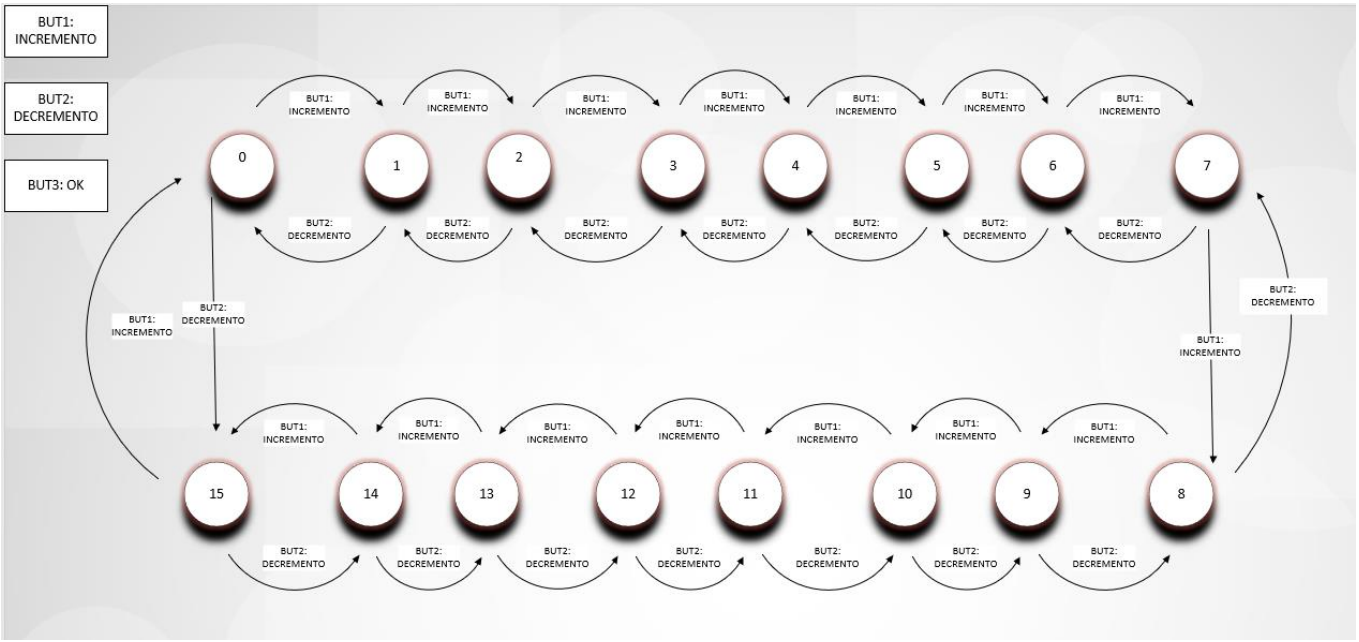


Figura 3. Diagrama de Transición de Estados del Módulo Contador

➢ **Módulo divisor:**

Para el divisor (Figura 4) se implementó la lógica mencionada anteriormente, donde constamos de 4 estados. Los cuales son:

- Carga
- Validación
- Cálculo
- Finalizado

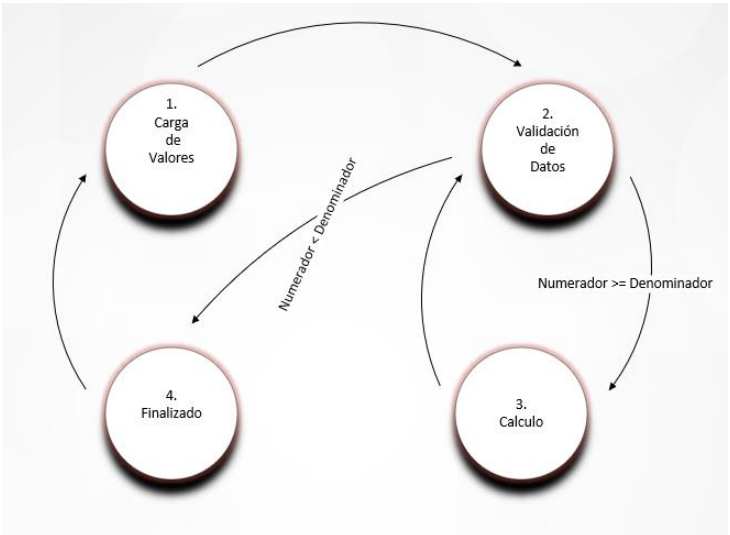


Figura 4. Diagrama de flujo de datos FSM divisor

La diferencia con el contador es que no necesitamos implementar botones, sino que se realiza con el flanco de subida del clk. Sus entradas son el clk(reloj), los reseteos sincrónico y asincrónico, el comienzo(start), los datos de entrada (numerador y denominador) y los datos de salida (cociente, resto y fin). Se definieron registros para almacenar el estado actual y siguiente de la FSM. También se definieron dos bloques always donde la funcionalidad de cada uno es:

- El primer bloque always se encarga de controlar la sincronización y el restablecimiento del circuito en función de las señales de reloj (clk) y reset (rst). Dentro de este bloque se verifica si la señal de reset (rst) está activa. Si es así, se asignan los valores iniciales a todas las variables internas, estableciendo el estado en carga y los registros en cero. Además, la señal finish se establece en cero. Si la señal de reset no está activa, se asignan los valores actuales de las variables internas a los registros de retención correspondientes. En resumen, este primer bloque always asegura que el circuito comience en un estado predefinido cuando se active la señal de reset (rst), y que las variables internas mantengan su valor actual cuando no haya un reset activo. Esto proporciona un mecanismo para inicializar y sincronizar el circuito correctamente.
- El segundo bloque always implementa la lógica del algoritmo de división, actualizando las variables internas en función del estado actual y las señales de entrada. Esto permite realizar los cálculos necesarios para la división y controlar el flujo del circuito hasta que se complete la operación.

- **Modulo multiplexor:**

Este módulo, como el mismo nombre lo dice, cumple la función de ser un multiplexor. Pero, ¿para que usamos un multiplexor? Lo usamos justamente para multiplexar lo que queremos mostrar. Es decir, nosotros en la FPGA, debemos mostrar el numerador, el denominador, el cociente y el resto, pero esto obviamente no se puede mostrar al mismo tiempo, por lo que un multiplexor de 2 bits (Figura 5) para elegir que queremos mostrar nos fue de muchísima utilidad. Al mismo le pasamos cada bit, de lo que queremos mostrar, y un selector. Entonces dependiendo el selector va a mostrar el numerador, el denominador, el cociente y el resto. Dejo aquí un pequeño grafico para que sea mas legible el entendimiento del mismo:

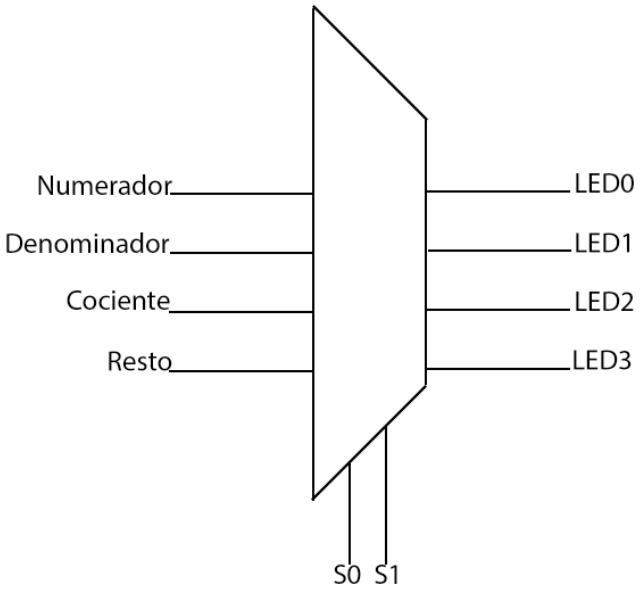


Figura 5. Multiplexor

- **Modulo top:**

En términos de implementación, es el módulo más largo. Este módulo, representado en la Figura 6, es el hace la llamada a los módulos anteriores y se hace la programación del algoritmo completo.

Figura 6. Diagrama de flujo de datos de FSM top

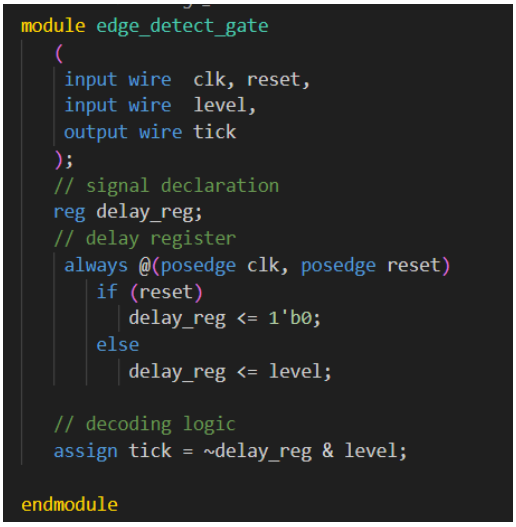
Un problema que se nos presentó durante la implementación de la misma fue que al apretar el botón en la FPGA para que el contador se incremente, ese incrementaba una gran cantidad de veces haciendo que no sea posible seleccionar el valor requerido, ya que el reloj de la FPGA oscila a 12MHz. Lo que significa que, si apretamos el botón 1 segundo, el contador avanzara millones de veces, por lo que sería imposible elegir el valor que queremos. Este problema se solucionó colocando dos módulos, uno llamado **debouncing** y otro llamado **edge_detect** , en la siguiente explicación se ve la resolución de este problema y básicamente el módulo completo:

Modulo edge_detect:

El módulo tiene las siguientes entradas y salidas:

- **clk**: Señal de reloj. (entrada)
- **reset**: Señal de reinicio. (entrada)
- **level**: Señal de nivel que se va a analizar. (entrada)
- **tick**: Salida que indica la detección de un flanco de subida en la señal de nivel. (salida)

donde podemos ver su código implementado en la figura 7:



```
module edge_detect_gate
   (
    input wire  clk, reset,
    input wire  level,
    output wire tick
   );
   // signal declaration
   reg delay_reg;
   // delay register
    always @(posedge clk, posedge reset)
       if (reset)
          delay_reg <= 1'b0;
       else
          delay_reg <= level;

   // decoding logic
   assign tick = ~delay_reg & level;

endmodule
```

Figura 7: Modulo Edge detect.

En resumen, el módulo **edge_detect_gate** detecta un flanco de subida en la señal de nivel utilizando un registro de retardo. Cuando se detecta el flanco de subida, la salida **tick** se activa. Este módulo es útil para resolver el problema de múltiples incrementos del contador que se mencionaron anteriormente. Al utilizar el módulo **edge_detect_gate** en conjunto con un mecanismo de eliminación de rebotes (debouncing) el cual explicaremos a continuación, se puede asegurar que el contador se incremente solo una vez por cada pulsación del botón, evitando incrementos múltiples no deseados debido a fluctuaciones o ruido en la señal del botón.

Module debouncing:

El módulo **db_fsm** implementa un mecanismo de eliminación de rebotes (debouncing) utilizando una máquina de estados finitos (FSM). Su objetivo es estabilizar una señal de entrada para evitar fluctuaciones o rebotes que puedan ocurrir al presionar un botón o cambiar un interruptor. Es decir, resuelve el problema de los rebotes en una señal de entrada, que ocurren cuando se presiona un botón o se cambia un interruptor y la señal fluctúa rápidamente en lugar de mantenerse estable.

El módulo cuenta con estados que definen el comportamiento del debouncing y utiliza un contador para generar un tick de tiempo. La FSM realiza transiciones de estado y actualiza la salida **db** cuando se detecta un estado estable en la señal de entrada.

El código del módulo top comienza aplicando el módulo debouncing a cada uno de los botones, luego invirtiendo los botones de la FPGA (Figura 8), ya que tienen la particularidad de que mandan una señal 1 cuando no son presionados y 0 cuando lo son. Como los módulos internos fueron implementados con una lógica inversa, los botones de entrada deben ser invertidos. Lo realizamos de la siguiente manera:

```
assign BTN[0] = ~xBTN[0];
assign BTN[1] = ~xBTN[1];
assign BTN[2] = ~xBTN[2];
assign BTN[3] = ~xBTN[3];
```

xBTN1, xBTN2, xBTN3 y xBTN4 (botones que se ingresan)

Figura 8. Inversión de los botones

Luego de hacer la inversión de los botones, aplicamos el module Edge_detect a cada uno de los botones. Siguiendo definimos los estados de la maquina de estado, el mismo tendrá los siguientes estados:

- Cargar Numerador: Sera el encargado del control de la carga del numerador.
- Cargar Denominador: Sera el encargado del control de la carga del denominador.

- Chequeo: en este estado se verifica que el denominador sea distinto de cero..
- Mostrar Cociente: estado encargado de mostrar el cociente de la división.
- Mostrar Resto: estado encargado de mostrar el resto de la división.

Siguiendo, se instanciaron dos módulos **contu** y **contd** utilizando el contador definido anteriormente **contu** es el contador para el numerador, **contd** es el contador para el denominador. Se definieron por separado para evitar problemas y ambigüedades a la hora de cargar el numerador y denominador. Estos módulos reciben la señal de reloj (**clk**), la señal de reinicio (**reset,** botón 3 de la FPGA), la señal de aumento (**up**, botón 1 de la FPGA), la señal de disminución (**down,** botón 2 de la FPGA), y proporcionan una cuenta para el numerador (**count_exi_num**) y una cuenta para el denominador (**count_exi_deno**). Estos contadores se utilizan para realizar operaciones aritméticas. En la figura 9 podemos ver los módulos **contu y contd.**

```
wire [3:0] count_exi_num;
contu cont_module_numerador (
    .clk(clk),
    .reset(BTNE[3]),
    .up(BTNE[1]),
    .down(BTNE[2]),
    .count(count_exi_num)
);

wire [3:0] count_exi_deno;
contd cont_module_denominador (
    .clk(clk),
    .reset(BTNE[3]),
    .up(BTNE[1]),
    .down(BTNE[2]),
    .count(count_exi_deno)
);
```

Figura 9. **Módulos contu y contd.**

Luego se instancio el módulo **div** (Figura 10) para realizar una operación de división. Este módulo recibe la señal de reloj (**clk**), la señal de reinicio (**rst**), la señal de inicio (**start**), la cuenta del numerador (**count_exi_num**), la cuenta del denominador (**count_exi_deno**), y proporciona el cociente (**cociente**), el resto (**resto**) y la señal de finalización (**finish**).

```
div div_module(
    .clk(clk),
    .rst(BTNE[3]),
    .start(start),
    .cociente(cociente),
    .numerador(count_exi_num),
    .resto(resto),
    .denominador(count_exi_deno),
    .finish(finish)
);
```

Figura 10. Modulo div.

Siguiendo, se instancio el módulo **mux_4_1** (Figura 11) para implementar un multiplexor. Este multiplexor recibe las cuentas del numerador (**conta_num**), las cuentas del denominador (**conta_den**), el cociente (**cociente**) y el resto (**resto**). También recibe una señal de selección (**Sel**) que determina qué entrada se selecciona y se asigna a la salida de los LEDs (**LED**).

```
mux_4_1 multiplexor_leds(
    .conta_num(count_exi_num),
    .conta_den(count_exi_deno),
    .cociente(cociente),
    .resto(resto),
    .Sel(sel),
    .LEDS(LED)
);
```

Figura 11. Modulo mux_4_1

Ahora se explicará la lógica de los dos bloques always, donde estos se utilizan para para controlar el estado del sistema y la transición entre estados.

El primer bloque **always** se activa en el flanco ascendente del reloj (**posedge clk**) o en el flanco ascendente del botón de reinicio (**posedge BTNE[3]**). Dentro de este bloque, se verifica si el botón de reinicio ha sido presionado (**BTNE[3]**). Si es así, se asignan los valores iniciales a los registros **state_reg**, **start_reg** y **sel_reg**. En caso contrario, se actualizan los registros con sus respectivos valores futuros (**state_next**, **start_next**, **sel_next**).

El segundo bloque **always** se activa cuando cualquiera de las señales en su lista de sensibilidad (**@***) cambia. Dentro de este bloque, se utiliza una estructura **case** para determinar el estado actual (**state_reg**) y realizar las transiciones de estado en función de las condiciones especificadas. Depende de donde se encuentre **state_reg** , se realiza la lógica correspondiente para cada caso. State_reg puede estar en distintos estados como: cargar_num, cargar_den, chequeo, mostrar_coc, mostrar_res.

Si estamos en el estado cargar_num y se aprieta el botón OK (Botón 0) para cambiar de estado, el siguiente estado será cargar_den

Si estamos en el estado cargar_den y se aprieta el botón OK (Botón 0) para cambiar de estado, el siguiente estado será chequeo.

Si estamos en el estado mostrar_coc y se aprieta el botón OK (Botón 0) para cambiar de estado, el siguiente estado será mostrar_res.

Si estamos en el estado mostrar_res y se aprieta el botón OK (Botón 0) para cambiar de estado, el siguiente estado será cargar_num.

Finalmente, se asignan las señales **sel_reg** y **start_reg** a las señales de salida **sel** y **start** respectivamente mediante la instrucción **assign**. Esto permite que los valores almacenados en los registros sean propagados a las salidas del módulo.

3. Conclusión

En conclusión, este informe integrador final ha abarcado el diseño e implementación de un divisor de cuatro bits de números enteros utilizando una FPGA y una FSM. Se ha seguido un algoritmo específico para realizar la operación de división entera y se han utilizado los recursos de la FPGA, junto con el lenguaje de descripción de hardware Verilog, para programar y probar el funcionamiento del divisor.

El algoritmo propuesto se basa en cargar los datos del numerador y denominador, verificar la viabilidad de la división, y realizar iteraciones sucesivas restando al numerador el valor del denominador hasta que el numerador sea menor al denominador. El resultado de la división se obtiene contando el número de iteraciones realizadas, y el resto se determina como el valor del numerador después de la última iteración.

Para llevar a cabo este algoritmo, se ha diseñado una FSM que representa el comportamiento del sistema. Se han establecido decisiones de diseño para garantizar la eficiencia y funcionalidad del circuito. Durante el proceso de desarrollo, se han enfrentado desafíos técnicos, como la sincronización del contador y la resolución de problemas relacionados con la frecuencia del reloj de la FPGA. Sin embargo, se han encontrado soluciones efectivas para superar estos obstáculos y lograr un diseño estable y consistente.

El resultado final es un divisor de cuatro bits que cumple con los requisitos planteados, permitiendo al usuario cargar los valores del numerador y denominador, realizar la división y mostrar el cociente y el resto a través de los LEDs de la FPGA. Se ha realizado una implementación robusta y se ha verificado su funcionamiento mediante pruebas de simulación y la utilización de un testbench.

4. Referencias

[1] P. P. Chu, FPGA Prototyping by Verilog Examples. Hoboken, New Jersey: John Wiley & Sons, Inc., 2008