

TAREA 2: DISEÑO Y VERIFICACIÓN DE UN DIVISOR SECUENCIAL BASADO EN SUMAS Y DESPLAZAMIENTOS

Bernardo Enguix Chordá, Marcos Ibáñez Fandos, Salvador Marí Selfa, Arnau Mora Gras, Julia Navarro Vicent, Carlos Villena Jiménez

INTRODUCCIÓN

La tarea consiste en diseñar y verificar un divisor binario con el objetivo de implementar directamente el ASM mediante systemVerilog y de ejercitar la realización de bancos de pruebas con systemVerilog con RCSG, cobertura funcional, clases, modelos de referencia, interfaces, bloques de reloj y aserciones.

Se podrá ver el código por completo en el repositorio de github enlazado aquí →

<https://github.com/ArnyminerZ/UPV-ISDIGI-DIVISORBINARIO>

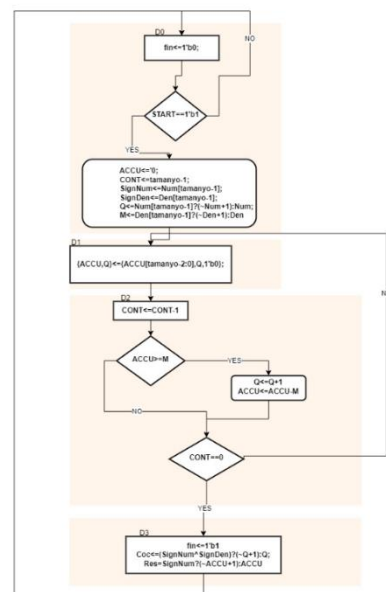
ESTRUCTURA

Diseño Etapa RTL:

- a. Diseño Componente Data-path
- b. Diseño Componente Control-path
- c. Descripción del sistema y verificación funcional

Diseño Verificación:

- a. Simulación. Compilación del sistema y simulación lógica
- b. Verificación lógica BÁSICA del diseño realizado
- c. Verificación intermedia y avanzada



En este bloque inicial se puede ver las señales externas de entrada y salida correspondientes al acceso al divisor descritas en la Fig.2.

```
module Divisor_Algoritmico #(
    // El tamaño en bits del divisor
    parameter integer tamanyo = 32,

    // Constantes de estado
    parameter S0 = 2'd0,
    parameter S1 = 2'd1,
    parameter S2 = 2'd2,
    parameter S3 = 2'd3
)(
    // ! Entradas ! \\
    input CLK, RSTa, Start,
    input logic [tamanyo-1:0] Num, Den,

    // ! Salidas ! \\
    output logic [tamanyo-1:0] Coc, Res,
    output logic Done
);
// Contenedor del estado
logic [1:0] state;

// Contenedor del valor de 2s complement que se está usando (ver README.md)
logic [tamanyo-1:0] mem, c2s;

// Contenedor del contador para el cociente
logic [tamanyo-1:0] q;

logic [tamanyo-1:0] posDen;

logic signNum, signDen;
```

También se conocen los parámetros internos del Divisor.

Se define un contenedor del estado, **state**, en el que tendremos 4 constantes ya que hay 4 bloques; **S0, S1, S2, S3**.

Se emplean dos parámetros para el de valor de complemento a dos; **mem** de 32 bits y **c2s**.

Para el cociente del contador se tienen las variables de 32 bits; **q** y **posDen**. También se definen las variables **signNum** y **signDen** que indicarán el signo del numerador y denominador.

Al principio de este fichero systemVerilog se implementa un FMS, que consiste en una celda altamente automatizada de Tecnologías de Grupos, que consiste de un grupo de estaciones de trabajo de procesos, interconectadas por un sistema automático de carga, almacenamiento y descarga de materiales.

CONTROL-PATH

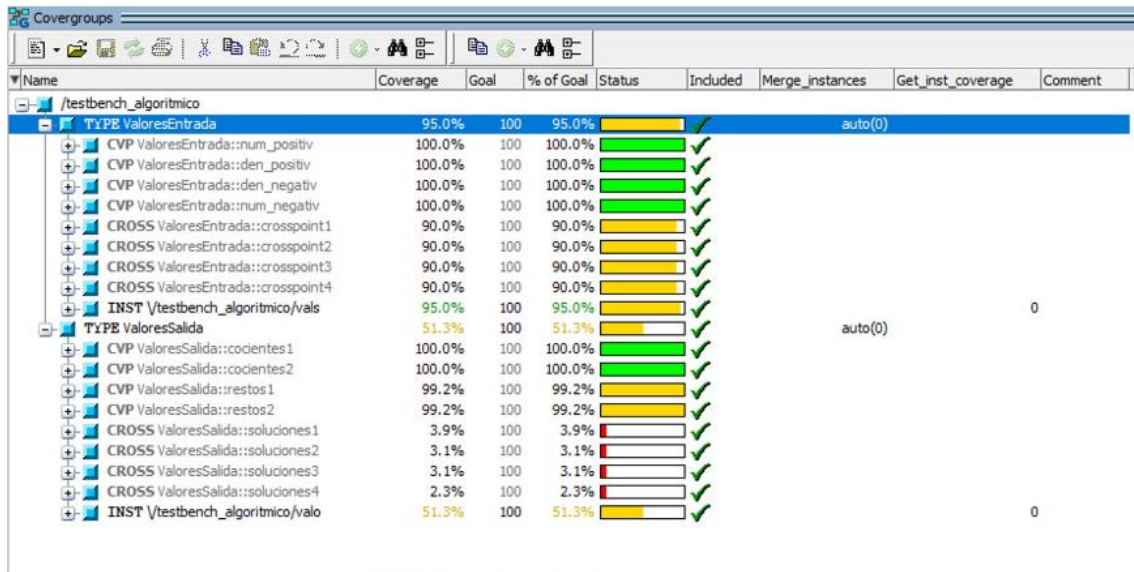
El control path organiza, administra y controla el estado. La señal de sensibilidad del control path será la propia entrada start que se va a alimentar en su etapa inicial.

DATA-PATH

El data path ejecuta todos los cambios en todas las variables existentes en el diseño.

1.2 Verificación del modelo RTL generado.

En la verificación se intenta obtener un grado de cobertura que garantice que el diseño es correcto o no.



Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/testbench_algoritmico								
TYPE ValoresEntrada	95.0%	100	95.0%					auto(0)
CVP ValoresEntrada::num_positiv	100.0%	100	100.0%					
CVP ValoresEntrada::den_positiv	100.0%	100	100.0%					
CVP ValoresEntrada::den_negativ	100.0%	100	100.0%					
CVP ValoresEntrada::num_negativ	100.0%	100	100.0%					
CROSS ValoresEntrada::crosspoint1	90.0%	100	90.0%					
CROSS ValoresEntrada::crosspoint2	90.0%	100	90.0%					
CROSS ValoresEntrada::crosspoint3	90.0%	100	90.0%					
CROSS ValoresEntrada::crosspoint4	90.0%	100	90.0%					
INST \testbench_algoritmico/vals	95.0%	100	95.0%					0
TYPE ValoresSalida	51.3%	100	51.3%					auto(0)
CVP ValoresSalida::cocientes1	100.0%	100	100.0%					
CVP ValoresSalida::cocientes2	100.0%	100	100.0%					
CVP ValoresSalida::restos1	99.2%	100	99.2%					
CVP ValoresSalida::restos2	99.2%	100	99.2%					
CROSS ValoresSalida::soluciones1	3.9%	100	3.9%					
CROSS ValoresSalida::soluciones2	3.1%	100	3.1%					
CROSS ValoresSalida::soluciones3	3.1%	100	3.1%					
CROSS ValoresSalida::soluciones4	2.3%	100	2.3%					
INST \testbench_algoritmico/valo	51.3%	100	51.3%					0

Cómo se aprecia en los crosspoints de las entradas, todos ellos tienen un coverage del 90% que garantiza un funcionamiento más que correcto.

Además del covergroup encargado de las entradas, el equipo ha decidido implementar un covergroup con trigger que se encarga de almacenar los cocientes y restos que dan las operaciones que realizadas. La esperanza en la combinatoria de este no era tan alta como en las entradas ya que no dependen puramente del equipo, por lo que han resultado con bastante poco coverage. Una posible causa puede ser los estados ilegales que plantean las divisiones con resto con un número tan limitado de bits, ya que hay algunas combinaciones que son francamente imposibles de conseguir. No obstante, todos los restos y cocientes posibles por separado han sido verificados.

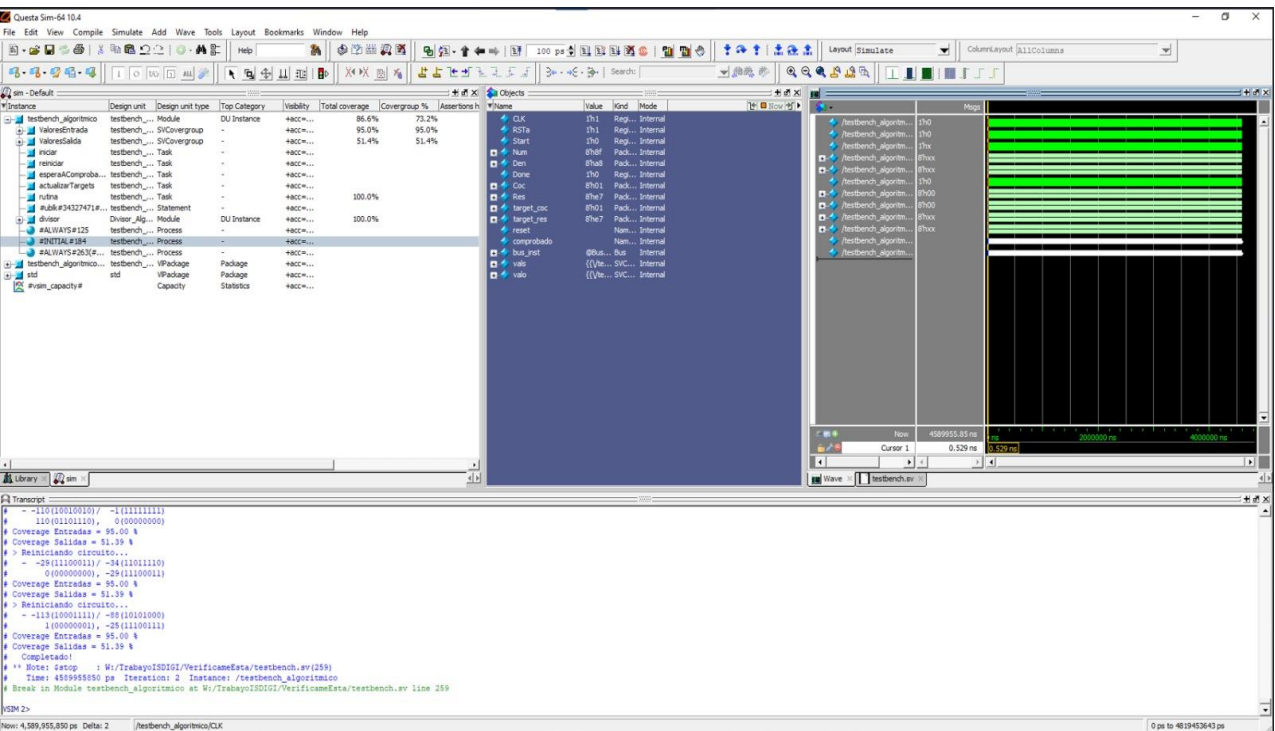
```
covergroup ValoresEntrada;
  coverpoint1: coverpoint Num {bins binsNumPos[('BIN_SIZE)/2] = {[0:('BIN_SIZE)/2-1]};}
  coverpoint2: coverpoint Den {bins binsDenPos[('BIN_SIZE)/2-1] = {[1:('BIN_SIZE)/2-1]};
  illegal_bins zero[1] = {0}; //denominador = 0 es un estado ilegal
  coverpoint3: coverpoint Den {bins binsDenNeg[('BIN_SIZE)/2] = {[-(('BIN_SIZE)/2)-1]};}
  coverpoint4: coverpoint Num {bins binsNumPos[('BIN_SIZE)/2] = {[-(('BIN_SIZE)/2)-1]};}
  coverpoint5: cross coverpoint1,coverpoint2; //combinatoria de numerador positivo y denominador pos
  coverpoint6: cross coverpoint1,coverpoint3; //combinatoria de numerador positivo y denominador neg
  coverpoint7: cross coverpoint4,coverpoint2; //combinatoria de numerador negativo y denominador pos
  coverpoint8: cross coverpoint4,coverpoint3; //combinatoria de numerador negativo y denominador neg
endgroup
covergroup ValoresSalida @(negedge Done);
  cocientes1: coverpoint Coc {bins binsCocPos[('BIN_SIZE)/2] = {[0:('BIN_SIZE)/2-1]};}
  cocientes2: coverpoint Coc {bins binsCocNeg[('BIN_SIZE)/2] = {[0:('BIN_SIZE)/2-1]};}
  restos1: coverpoint Res {bins binsResPos[('BIN_SIZE)/2] = {[0:('BIN_SIZE)/2-1]};}
  restos2: coverpoint Res {bins binsResNeg[('BIN_SIZE)/2] = {[0:('BIN_SIZE)/2-1]};}
  soluciones1: cross cocientes1,restos1; //combinatoria cocientes positivos y restos positivos
  soluciones2: cross cocientes1,restos2; //combinatoria cocientes positivos y restos negativos
  soluciones3: cross cocientes2,restos1; //combinatoria cocientes negativos y restos positivos
  soluciones4: cross cocientes2,restos2; //combinatoria cocientes negativos y restos negativos
endgroup
```

Aquí se observa la creación de los coverages que se han utilizado, como se puede apreciar en el coverpoint2 hay una bin definida como ilegal. Esta es la que corresponde al número 0, ya que el denominador no puede ser cero en una división. Se está ante un estado ilegal que no se puede permitir bajo ningún concepto, por lo que no se cuenta con ella.

En cuanto a las assertions, el chequeo realizado es el siguiente →

*Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression	Included
/testbench_algoritmico/divisor/reseteado	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	-	✓
/testbench_algoritmico/futuro/med_167	Immediate	SVA	on	0	1	-	-	-	-	-	off	-	✓
/testbench_algoritmico/futuro/med_169	Immediate	SVA	on	0	1	-	-	-	-	-	off	-	✓
/testbench_algoritmico/publi#34327471#262/med_273	Immediate	SVA	on	0	1	-	-	-	-	-	off	-	✓
/testbench_algoritmico/publi#34327471#262/med_274	Immediate	SVA	on	0	1	-	-	-	-	-	off	-	✓

2. ETAPA LÓGICA. COMPILACIÓN DEL DIVISOR Y SIMULACIÓN LÓGICA.



Hemos realizado el gate-level simulation y como podemos observar se ha simulado todo sin ningún tipo de problema.

Para realizar el análisis temporal del divisor algorítmico utilizaremos la herramienta TimeQuest TimingAnalyser una vez hayamos compilado el proyecto.

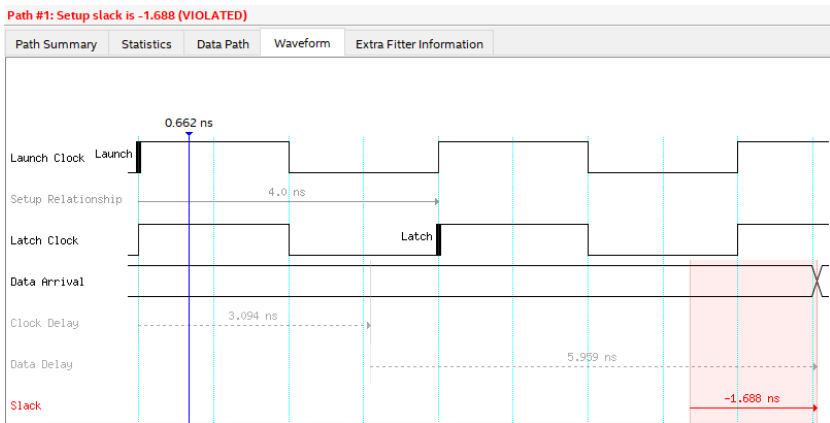
En primer lugar comprobaremos que relojes detecta. Podemos ver que detecta una señal de reloj denominada CLK con las siguientes características:

	Fmax	Restricted Fmax	Clock Name
1	175.81 MHz	175.81 MHz	CLK

La frecuencia objetivo es bastante elevada. Por ello modificamos este valor definiendo el periodo como 4.000 ns y 2.000 ns de caída (falling). Así obtenemos la nueva frecuencia objetivo.

El siguiente paso es sacar la frecuencia máxima de operación. Para ver los problemas de márgenes de activación haremos un report del setup, o podríamos hacer un report y ver cuál es el peor caso con los 10 peores casos.

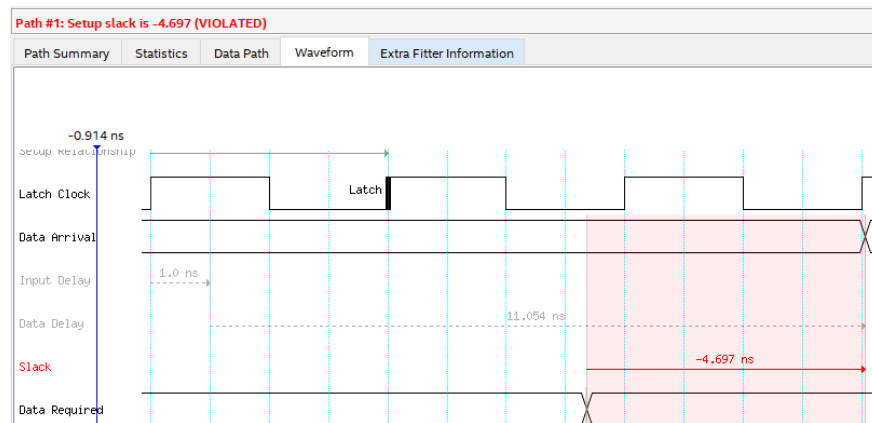
	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-1.688	num_c2s[0]	num_c2s[31]	CLK	CLK	4.000	0.273	5.959
2	-1.674	state.S0	num_c2s[31]	CLK	CLK	4.000	0.275	5.947
3	-1.638	state.S0	num_c2s[31]	CLK	CLK	4.000	0.275	5.911
4	-1.568	state.S0	num_c2s[31]	CLK	CLK	4.000	0.275	5.841
5	-1.500	state.S0	num_c2s[31]	CLK	CLK	4.000	0.275	5.773
6	-1.481	num_c2s[0]	q[31]	CLK	CLK	4.000	0.328	5.807
7	-1.481	num_c2s[0]	q[30]	CLK	CLK	4.000	0.328	5.807
8	-1.481	num_c2s[0]	q[29]	CLK	CLK	4.000	0.328	5.807
9	-1.481	num_c2s[0]	q[28]	CLK	CLK	4.000	0.328	5.807
10	-1.481	num_c2s[0]	q[27]	CLK	CLK	4.000	0.328	5.807



Si queremos introducir información sobre qué ocurre desde el exterior hasta los puertos de entrada del divisor algorítmico, tendremos más información de más paths.

En Set Input Delay añadimos un retardo de 1 ns y listamos todos los puertos de entrada (Num, Den, Start, RSTa y CLK).

	Fmax	Restricted Fmax	Clock Name
1	114.98 MHz	114.98 MHz	CLK

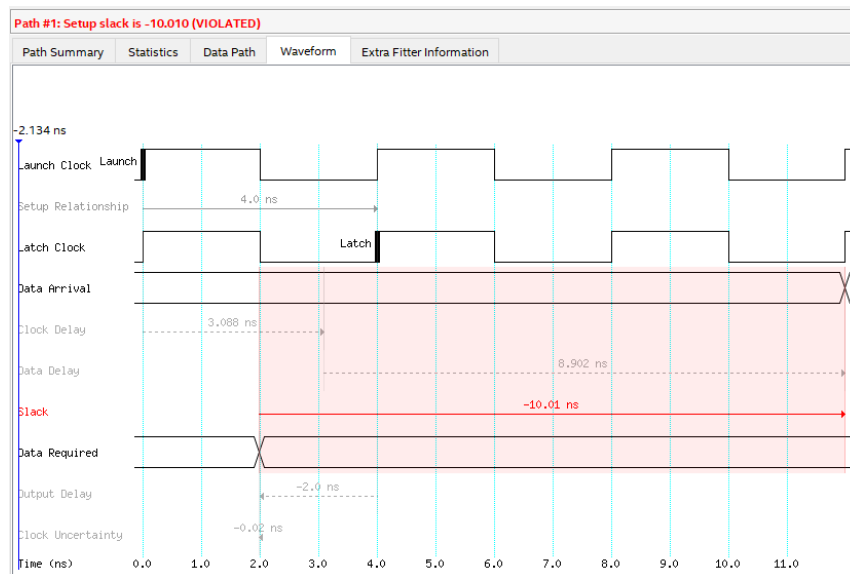


	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-4.697	Den[0]	den_abs[25]	CLK	CLK	4.000	3.359	11.054
2	-4.632	Den[0]	den_c2s[27]	CLK	CLK	4.000	2.981	10.611
3	-4.571	Den[0]	den_abs[15]	CLK	CLK	4.000	2.971	10.540
4	-4.565	Den[0]	den_abs[23]	CLK	CLK	4.000	2.970	10.533
5	-4.558	Den[0]	den_c2s[20]	CLK	CLK	4.000	2.968	10.524
6	-4.459	Num[1]	num_c2s[31]	CLK	CLK	4.000	3.359	10.816
7	-4.436	Den[0]	den_abs[11]	CLK	CLK	4.000	2.971	10.405
8	-4.410	Den[0]	den_c2s[29]	CLK	CLK	4.000	2.973	10.381
9	-4.394	Num[2]	num_c2s[31]	CLK	CLK	4.000	3.359	10.751
10	-4.333	Den[0]	den_abs[8]	CLK	CLK	4.000	2.971	10.302

En Set Output Delay añadimos un retardo de 2 ns y listamos todos los puertos de salida (Coc, Res y Done).

	Fmax	Restricted Fmax	Clock Name
1	71.38 MHz	71.38 MHz	CLK

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-10.010	Coc[23]~reg0	Coc[23]	CLK	CLK	4.000	-3.088	8.902
2	-9.931	Coc[17]~reg0	Coc[17]	CLK	CLK	4.000	-3.088	8.823
3	-9.865	Coc[26]~reg0	Coc[26]	CLK	CLK	4.000	-3.088	8.757
4	-8.718	Coc[11]~reg0	Coc[11]	CLK	CLK	4.000	-3.091	7.607
5	-8.200	Res[31]~reg0	Res[31]	CLK	CLK	4.000	-3.505	6.675
6	-7.930	Res[7]~reg0	Res[7]	CLK	CLK	4.000	-3.075	6.835
7	-7.856	Res[30]~reg0	Res[30]	CLK	CLK	4.000	-3.505	6.331
8	-7.743	Done~reg0	Done	CLK	CLK	4.000	-3.092	6.631
9	-7.389	Res[25]~reg0	Res[25]	CLK	CLK	4.000	-3.505	5.864
10	-7.206	Res[20]~reg0	Res[20]	CLK	CLK	4.000	-3.073	6.113



Una vez realizado esto es conveniente escribir todas estas constraints sobre el fichero y volver a Quartus para volver a compilar pero esta vez teniendo en cuenta los atributos que hemos añadido. Además a la hora de simular especificamos la opción Speed, con el Timing-Driven Synthesisys activado.

Una vez realizada esta segunda compilación comprobamos que el clock coincide con el anterior y buscamos la frecuencia máxima de operación:

	Fmax	Restricted Fmax	Clock Name
1	123.66 MHz	123.66 MHz	CLK

3. DISEÑO DEL DIVISOR SEGMENTADO

3.1 Realización del código RTL

El divisor segmentado es la implementación de un divisor cualquiera como el planteado anteriormente, en donde el divisor segmentado procesa todas las operaciones necesarias de la división como el algorítmico, pero en un único ciclo de reloj, en vez de en varios ciclos de reloj.

Por lo que nos encontraremos con una mejor no de latencia sino de eficiencia y cadencia. En el divisor algorítmico nos encontrábamos con una cadencia igual a la latencia (1) y ahora esto ha cambiado con mejora en rendimiento (eficiencia).

Realizaremos un proceso de cambio de multiciclo (Algorítmico) a pipeline (Segmentado).

Para ello haremos uso de la función 'generate' la cual permite que se hagan múltiples acciones directamente, en cuya función se usará el bucle for para generar los 32 módulos de división (con flip flops) que se necesita para realizar la división de forma completa en ese ciclo de reloj.

Dentro del divisor segmentado instanciaremos un programa auxiliar que ayuda a realizar todo el proceso del cálculo necesario para meter dentro del bucle for.

Su estructura es muy parecida a la implementada en el divisor algorítmico:

```
module Aux_Segmentado #(
    parameter integer tamanyo = 32
) (
    // ! Entradas ! \
    input logic CLK, RSTa, Start,          // Declaramos las variables lógicas de reloj, reset y comienzo de la operación de división
    input logic SignNum, SignDen,          // Declaramos las variables de único bit de los signos del Denominador(Den) y Numerador(N)
    input logic [tamanyo-1:0] Q, M, ACCU,    // Declaramos las variables (Q,M,ACCU) de tamaño a elegir con la definición "LAST_BIT"

    // ! Salidas ! \
    output logic [tamanyo-1:0] Q_out, M_out, ACCU_out, // Declaramos las variables de salidas de (Q,M,ACCU) de igual tamaño a las entradas
    output logic SignNum_out, SignDen_out,          // Declaramos las variables de salida de los signos del numerador y denominador
    output logic Done                               // Declaramos la variables de salida(Done) que actúa como valor de Start a partir del
);

logic [tamanyo-1:0] ACCU_int, Q_int; // Declaramos los registros internos del acumulador(ACCU) y del cociente(Q), siempre de igual tamaño

assign {ACCU_int, Q_int} = {ACCU[tamanyo-2:0],Q,1'b0};

always_ff @(posedge CLK, negedge RSTa) begin
    // Para reiniciar el módulo, borramos el valor de todas las salidas
    if (!RSTa)
        {Q_out, M_out, ACCU_out, SignNum_out, SignDen_out, Done} = '0; // Llenamos a cero todas las variables si se activa la señal de RESET
    else begin // Si no se pulsa el reset y el reloj Actúa -->
        M_out <= M; // El valor del resto de la división queda registrada en la salida(M_out)
        SignNum_out <= SignNum; // Las salidas de los signos cogen ahora los valores previos/iniciales que tenían
        SignDen_out <= SignDen; //

        Q_out <= (ACCU_int >= M) ? Q_int + 1 : Q_int; // Si (ACCU_int>=M) se le asignará a la salida de Q(Q_out) un valor más al valor de Q
        ACCU_out <= (ACCU_int >= M) ? ACCU_int - M : ACCU_int; // Si (ACCU_int>=M) le asignamos a la salida del acumulador(ACCU_out) justo el resto
        Done <= Start; // Guardamos en Done la señal de start, y ahora Done actuará como el propio Start
    end
end

endmodule
```

Ahora en cuanto al diseño del segmentado, se le declara las variables que aparecen en la siguiente imagen →

```

`include "Aux_Segmentado.sv"

module Divisor_Segmentado #(
    parameter integer tamanyo = 32
) (
    // Declaramos aquí entradas y salidas -->

    // ! Entradas -->
    input CLK , RSTa , Start , // Declaramos la entrada de reloj , el Reset high lvl y la entrada higg lvl de iniciac
    input logic [tamanyo-1:0] Num , Den , // Declaramos las entradas del numerador(Num) y del denominador (Den) de 32
    // ! Outputs -->
    output logic Done , // Declaramos la salida Done para ver cuando justo acaba de hacer la división
    output logic [tamanyo-1:0] Coc , Res // Declaramos las salidas del cociente (Coc)
                                     // y del Resto (Res) del resultado de la división entre
                                     // el numerador y el divisor (32 bits también)
);

localparam etapas=tamanyo; //2**tamanyo;

logic [etapas-1:0][tamanyo-1:0] ACCU, Q, M; // Declaramos el acumulador, el contador del cociente y el del resto
logic [etapas-1:0] SignNum, SignDen, Done_mem; // Declaramos los array del signo del numerador, denominador y el esta

/*
Queremos realizar un Divisor Segmentado, es decir un divisor algorítmico pero que realice dicha
operación completa por ciclo de reloj , en vez de cada parte de la operación por ciclo de reloj.

Este divisor segmentado lo realizaremos haciendo uso de un bucle 'for' para que cada vez que se quiera hacer
la operación se haga en dicho golpe de reloj cada paso hasta completar el resultado completo de la división.

Para ello, también implementamos la función generate que ayuda a la implementación de varias acciones de forma úni
Es ahí donde atacaremos con los bucles 'for'.

```

Una vez declarada las variables que vamos a usar, pasamos a la declaración de la función 'generate' y su propia variable que se usara para desplazarnos entre módulos dentro del bucle for → (genvar i).

En cuanto a la implementación en el bucle, tendremos en cuenta dos módulos principales, que son el inicial (Cuando i=0) y el final (cuando i=tamaño=32=.....) en los cuales se procede de forma diferente entre ellos y entre los módulos {i=1,2,3,...,31}.

Este proceso lo presentamos de la siguiente forma →

Para estado i=0 →

```

generate
for(i = 0; i<(etapas+1) ; i = i+1) // Empezamos con i=0 hasta que i llegue como máximo a 32, va incrementando el bit de 1 en 1 (i=i+1)
begin :generador
    // Como se van a generar 32 módulos, usaremos un case/default para realizarlo, donde solo
    // se llegarán a declarar los módulos 0 y 32 de forma directa y en el default estarán los
    // 30 módulos restantes ya que estos dependen directamente del módulo anterior y operan de igual forma
    // (Los únicos módulos en los que se procede de forma diferente son en el primero(0) y último(tamanyo) ).

    case(i) // En caso de estar en módulo 'i' se realiza las siguientes acciones.

    0: // Módulo en cuanto la cuenta está sin empezar y justo se Activa la señal de cmoienzo.
        Aux_Segmentado #(
            .tamanyo(tamanyo)
        ) Comienzo_Division ( // Declaramos la primera instancia del divisor auxiliar que permite
                                // iniciar la cadena de sumadores.
                                .CLK(CLK), // Conectamos el CLK de la instancia al del módulo.
                                .RSTa(RSTa), // Conectamos el RSTa de la instancia al del módulo.
                                .Start(Start), // Conectamos el Start de la instancia al del módulo.
                                .SignNum(Num[tamanyo-1]), // Conectamos SignNum con el valor de Num y solo buscamos para el bit más significati
                                .SignDen(Den[tamanyo-1]), // De igual forma que la anterior pero para el signo del denominador .
                                .ACCU('0), // El acumulador al principio está a cero total ya que no se ha realizado de momento
                                .Q(Num[tamanyo-1] ? (~Num+1) : Num), //
                                .M(Den[tamanyo-1] ? (~Den+1) : Den), //
                                .ACCU_out(ACCU[i]), // La salida del ACCU está conectada con el valor que tenga de entrada para el estado
                                .Q_out(Q[i]), // Q_out coge la salida de Q a instancia de (i = 0)
                                .M_out(M[i]), // M_out se conecta a la propia entrada en el estado inicial (i = 0)
                                .SignNum_out(SignNum[i]), // La salida del signo en Num y Den coge los valor de las entradas para (i = 0)
                                .SignDen_out(SignDen[i]),
                                .Done(Done_mem[i]) // El valor que se guarda en Done va ha ser el valor que tenga asignado a primer mome
                            );

    etapas: // Módulo en cuanto la cuenta llega a su fin y se completa la división, aquí asignamos el valor final -->
        always_ff @(posedge CLK, negedge RSTa) // Declaramos un bloque procedurar always_ff en el que se activa a ciclo de reloj ascende
        begin
            // o cuando se active el reseteo de la operación.
            if(RSTa) // Si se quiere resetear -->

```

Para estado i=tamaño=etapas →

```

etapas: // Módulo en cuanto la cuenta llega a su fin y se completa la división, aquí asignamos el valor final -->
always_ff @(posedge CLK, negedge RSta) // Declaramos un bloque procedural always_ff en el que se activa a ciclo de reloj ascend
begin
    if(!RSta) // Si se quiere resetear -->
    begin
        Coc <= '0; // Llenamos el cociente de ceros (ya que se inicia toda la operación se queda a full ceros)
        Res <= '0; // Ocurre lo mismo con los bits del 'resto'
        Done <= 1'b0; // Ponemos a cero el output de 'trabajo realizado'
    end
    else // Si no se activa el reset, sino que la operación va a quedar como concluida -->
    begin
        Coc <= (SignNum[i-1]^SignDen[i-1]) ? (~Q[i-1]+1) : Q[i-1]; // Si se cumple que ((SignNum[i-1] XOR SignDen[i-1]) == 1
        Res <= (SignNum[i-1]) ? (~ACC[i-1]+1) : ACC[i-1]; // Si se cumple que el signo está activo en ese instante,
        Done <= Done_mem[i-1]; // Actualizamos el valor del start con el valor que se haya guardado en el último módul
    end
end

default: // Con el default entramos en cualquier módulo diferente del inicial(0) y el final(etapas)
Aux_Segmentado #(
    .tamanyo(tamanyo)
) Siguiendo_Division [0] // Declaramos la segunda instancia del divisor auxiliar que permite
// continuar la cadena de sumadores desde i=1 hasta i=31.
CLK(CLK) // Conectamos el CLK de la instancia al del módulo

```

Para estado 'default' el que incluye los estados restantes que falta entre medias de $i=0$ y $i=\max$
→

```

default: // Con el default entramos en cualquier módulo diferente del inicial(0) y el final(etapas)
Aux_Segmentado #(
    .tamanyo(tamanyo)
) Siguiendo_Division [0] // Declaramos la segunda instancia del divisor auxiliar que permite
// continuar la cadena de sumadores desde i=1 hasta i=31.
CLK(CLK), // Conectamos el CLK de la instancia al del módulo
RSta(RSta), // Conectamos el RSta de la instancia al del módulo
Start(Done_mem[i-1]), // Conectamos el Start de la instancia al valor anterior asignado en Done_mem (ya que Done_mem
SignNum(SignNum[i-1]), // Como estamos en los módulos comprendidos en [1, LAST_BIT], el valor de signo actual será otro
SignDen(SignDen[i-1]), // Igual ocurrirá con el signo del denominador
ACC[ACC[i-1]), // Igual ocurre con el bit del acumulador
Q(Q[i-1]), // Guardamos Q como su valor previo
M(M[i-1]), // Guardamos M como su valor previo
ACC_out(ACC[i]), // En cambio, para la salida se guardará el que justo vayamos a utilizar para el siguiente mód
Q_out(Q[i]), // La salida del registro Q se le asignará el valor justo que tenga en ese preciso módulo (Que
M_out(M[i]), // Lo mismo ocurre con la salida del registro M
SignNum_out(SignNum[i]), // Las salidas de los signos tienen el valor actual
SignDen_out(SignDen[i]), // Las salidas de los signos tienen el valor actual
Done(Done_mem[i]) // Guardamos el último valor de lo que haya guardado en Done_mem
];

endcase // Terminamos el bucle case/default
end
endgenerate // Terminamos la función generate
endmodule

```

3.2 Verificación funcional.

En cuanto a la verificación del diseño del divisor segmentado hemos empleado la misma estructura que para el algorítmico, solo que lo hacemos con módulos, es decir, declaramos un testbench que sirva para todos los casos, entonces, en cuanto al segmentado solo es necesario declarar el nombre del módulo, incluir en dicho archivo verilog el nombre del testbench principal y así se realiza todo más rápido.

En cuanto al segmentado, quedaría tal que así →

```

Aux_Segmentado.sv  Divisor_Segmentado.sv  testbench_segmentado.sv  testbench.sv
VerificameEsta > testbench_segmentado.sv > ...
1
2 // Importamos el archivo del módulo
3 `include "../Divisor_Segmentado.sv" Arnau Hora, hace 3 horas • Generalizado testbenchs
4
5 // Declaramos el nombre del módulo
6 `define MODULO Divisor_Segmentado
7
8 // Declaramos el nombre que asignar al testbench
9 `define NOMBRE_TESTBENCH testbench_segmentado
10
11 // Para controlar parámetros genéricos como DEBUG, el objetivo del coverage,
12 // o el tamaño de bits, modificar testbench.sv
13
14 `include "testbench.sv"
15

```

Y el testbench principal sería de tal forma →

```

9 // Se puede definir debug para obtener registros mas detallados
10 `define DEBUG
11
12 // Qué valores deben ser probados?
13 // Numerador positivo, denominador positivo
14 `define TEST_NUM_POS_DEN_POS
15 // Numerador positivo, denominador negativo
16 `define TEST_NUM_POS_DEN_NEG
17 // Numerador negativo, denominador positivo
18 `define TEST_NUM_NEG_DEN_POS
19 // Numerador negativo, denominador negativo
20 `define TEST_NUM_NEG_DEN_NEG
21
22 // El objetivo de coverage
23 `define TB_COVERAGE 90
24
25 // Objetivo de coverage secundario, intentaremos probar cuantas mas posibles soluciones mejor sabiendo que hay algunas que no son posibles
26 `define TB_COVERAGE2 70 // Arnau Mora, hace 3 horas • Nuevo testbench
27
28 // El tamaño de bits que probar
29 `define BIT_SIZE 8 //probamos con ocho bits para que los coverpoints se encuentren dentro del margen que puede manejar esta construccion
30
31
32 // * NO CAMBIAR
33 `define LAST_BIT `BIT_SIZE-1
34 // TODO: Debería adaptarse a TEST_*
35 `define BIN_SIZE 2**`BIT_SIZE
36 // * FIN NO CAMBIAR
37
38
39 // La clase bus nos proporciona las constraints que nos permiten
40 // detallar qué queremos comprobar.
41 // Arnau Mora, hace 3 horas | 1 author (Arnau Mora)
42 class Bus;
43 randc logic signed [`LAST_BIT:0] num;
44 randc logic signed [`LAST_BIT:0] den;
45
46 // Para limitar a sólo denominadores positivos o negativos
47 constraint num_positivo {num[`LAST_BIT] == 1'b0;}
48 constraint num_negativo {num[`LAST_BIT] == 1'b1;}
49
50 // Para limitar a sólo numeradores positivos o negativos
51 constraint den_positivo {den[`LAST_BIT] == 1'b0;}
52 constraint den_negativo {den[`LAST_BIT] == 1'b1;}
53
54 // Como bien sabemos denominador nulo implica un estado ilegalísimo ya que no podemos dividir entre 0, evitando así uno de los estados mas i
55 constraint den_nozero {den != 0;}
56
57 // Para limitar a resultados sin residuo
58 constraint div_exactaaa {num%den == 0;}
59
60 // Limitamos el valor máximo, para dejar sitio para el signo. No
61 // desactivar esta constraint.
62 // Nota:
63 // - 2^15 = 32768 -> 16 bits
64 // - 2^31 = 2147483648 -> 32 bits
65 // constraint lim_grandeee {(abs(num)<32768) && (abs(den)<32768);}
66 endclass
67
68 // No definimos un nombre específico para el testbench, usamos el definido en
69 // testbench_algoritmico.sv y testbench_segmentado.sv para generalizar el testbench
70 // y no tener duplicados
71 module `NOMBRE_TESTBENCH;
72
73 logic CLK, RStA, Start;
74 logic signed [`LAST_BIT:0] Num, Den;
75 logic Done;
76 logic signed [`LAST_BIT:0] Coc, Res, target_coc, target_res;
77
78 event reset;
79 event comprobado;
80
81 covergroup ValoresEntrada;
82 num_positivo: coverpoint Num {bins binsNumPos[(`BIN_SIZE)/2] = {[0:((`BIN_SIZE)/2)-1]};
83 den_positivo: coverpoint Den {bins binsDenPos[(`BIN_SIZE)/2-1] = {[1:((`BIN_SIZE)/2)-1]};
84 den_negativo: coverpoint Den {bins binsDenNeg[(`BIN_SIZE)/2] = {[1:((`BIN_SIZE)/2)-1]};
85 num_negativo: coverpoint Num {bins binsNumNeg[(`BIN_SIZE)/2] = {[1:((`BIN_SIZE)/2)-1]};

```

```

87 crosspoint1: cross num_positiv,den_positiv; //combinatoria de numerador positivo y denominador positivo
88 crosspoint2: cross num_positiv,den_negativ; //combinatoria de numerador positivo y denominador negativo
89 crosspoint3: cross num_negativ,den_positiv; //combinatoria de numerador negativo y denominador positivo
90 crosspoint4: cross num_negativ,den_negativ; //combinatoria de numerador negativo y denominador negativo
91 endgroup
92 covergroup ValoresSalida @(negedge Done);
93 cocientes1: coverpoint Coc {bins binsCocPos[(`BIN_SIZE)/2] = {[0:(`BIN_SIZE)/2)-1]};}
94 cocientes2: coverpoint Coc {bins binsCocNeg[(`BIN_SIZE)/2] = {[0:(`BIN_SIZE)/2)-1]};}
95 restos1: coverpoint Res {bins binsResPos[(`BIN_SIZE)/2] = {[0:(`BIN_SIZE)/2)-1]};}
96 restos2: coverpoint Res {bins binsResNeg[(`BIN_SIZE)/2] = {[0:(`BIN_SIZE)/2)-1]};}
97 soluciones1: cross cocientes1,restos1; //combinatoria cocientes positivos y restos positivos
98 soluciones2: cross cocientes1,restos2; //combinatoria cocientes positivos y restos negativos
99 soluciones3: cross cocientes2,restos1; //combinatoria cocientes negativos y restos positivos
100 soluciones4: cross cocientes2,restos2; //combinatoria cocientes negativos y restos negativos
101 endgroup
102 // Declaración de objetos
103 Bus bus_inst;
104 ValoresEntrada vals;
105 ValoresSalida valo;
106
107 // Declaración de módulos
108 // Nota: el nombre del módulo instanciado se define en testbench_algorítmico.sv y
109 // testbench_segmentado.sv.
110 `MODULE #(
111     .tamanyo(`BIT_SIZE)
112 ) divisor (
113     .CLK(CLK),
114     .RSTa(RSTa),
115     .Start(Start),
116     .Num(Num),
117     .Den(Den),
118
119     .Coc(Coc),
120     .Res(Res),
121     .Done(Done)
122 );
123

```

En esta tercera imagen, es donde se declara el nombre del módulo al cual queremos correr, que puede ser el algorítmico o el segmentado.

A partir de ahí es mucho mejor ver el diseño de forma manual para no sobrecargar de fotos la memoria.

En cuanto al rango cubierto de coverage es el que mostramos a continuación →

/testbench_segmentado					
TYPE ValoresEntrada	73.24%	100	95.00%		auto(0)
CVP ValoresEntrada::num_positiv	95.00%	100	100.00%		
CVP ValoresEntrada::den_positiv	100.00%	100	100.00%		
CVP ValoresEntrada::den_negativ	100.00%	100	100.00%		
CVP ValoresEntrada::num_negativ	100.00%	100	100.00%		
CROSS ValoresEntrada::crosspoint1	90.00%	100	90.00%		
CROSS ValoresEntrada::crosspoint2	90.00%	100	90.00%		
CROSS ValoresEntrada::crosspoint3	90.00%	100	90.00%		
CROSS ValoresEntrada::crosspoint4	90.00%	100	90.00%		
INST Vtestbench_segmentado/vals	95.00%	100	95.00%		0
TYPE ValoresSalida	51.48%	100	51.48%		auto(0)
CVP ValoresSalida::cocientes1	100.00%	100	100.00%		
CVP ValoresSalida::cocientes2	100.00%	100	100.00%		
CVP ValoresSalida::restos1	100.00%	100	100.00%		
CVP ValoresSalida::restos2	99.21%	100	99.21%		
CROSS ValoresSalida::soluciones1	3.93%	100	3.93%		
CROSS ValoresSalida::soluciones2	3.16%	100	3.16%		
CROSS ValoresSalida::soluciones3	3.16%	100	3.16%		
CROSS ValoresSalida::soluciones4	2.38%	100	2.38%		
INST Vtestbench_segmentado/valo	51.48%	100	51.48%		0

Observamos los diferentes crosspoints, coverpoints realizados y testeados. Al igual que con el divisor algorítmico hemos obtenido una amplia cobertura del 90 por ciento de los casos posibles que nos garantiza ampliamente un funcionamiento idóneo. Además de esto aquí también hemos querido comprobar por nuestra cuenta la combinatoria entre los restos obtenidos y los cocientes obteniendo en este resultados muy similares al otro divisor llegando a las mismas conclusión.

En cuanto a las assertions, el chequeo realizado es el siguiente →

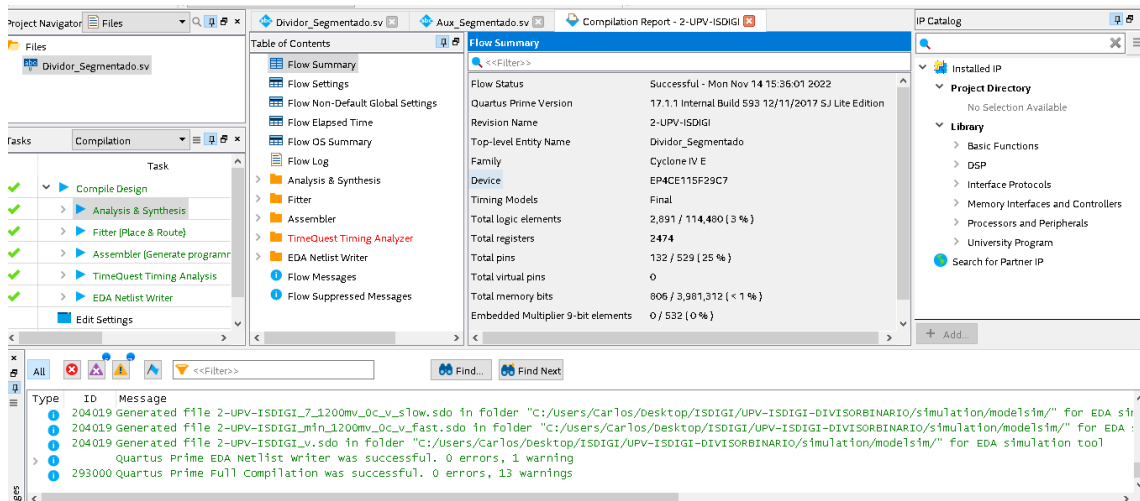
Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression	Included
./testbench_segmentado/divisor/resetado	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	-	✓
./testbench_segmentado/rutina/med_167	Immediate	SVA	on	0	1	-	-	-	-	0	off	-	✓
./testbench_segmentado/rutina/med_169	Immediate	SVA	on	0	1	-	-	-	-	0	off	-	✓
./testbench_segmentado/publi#105492979#262/med_273	Immediate	SVA	on	0	1	-	-	-	-	0	off	-	✓
./testbench_segmentado/publi#105492979#262/med_274	Immediate	SVA	on	0	1	-	-	-	-	0	off	-	✓

En la captura anterior, podemos observar las aserciones que han sido utilizadas para detectar posibles errores en el código. Como podemos observar han sido todo un éxito ya que ninguna de ellas ha fallado. Entre ellas podemos encontrar una aserción concurrente que comprueba que el reinicio se ejecute correctamente, por otro lado tenemos 4 aserciones inmediatas, dos de ellas comprueban la aleatorización de num y den y las otras dos comparan los resultados de tarjets con cociente y resto.

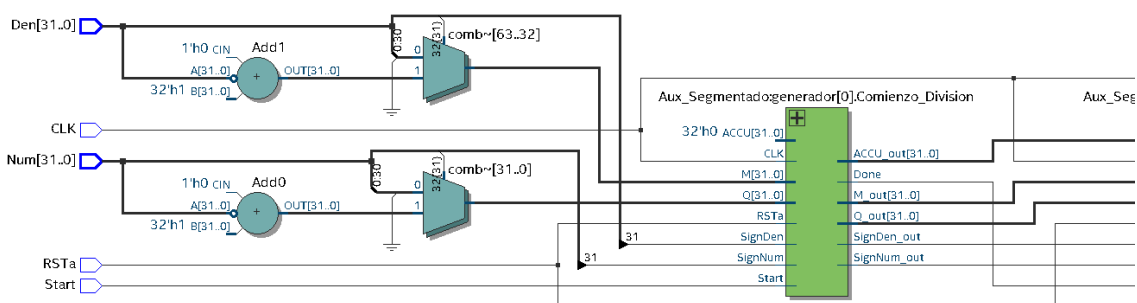
3.3 Compilación del diseño

El diseño del código planteado compila perfectamente, a excepción de 13 warnings no problemáticos, ya que son del uso de procesadores, de no asignaciones de pines en cuanto a input/output y demás que no se quitarán hasta que lo implementemos en placa.

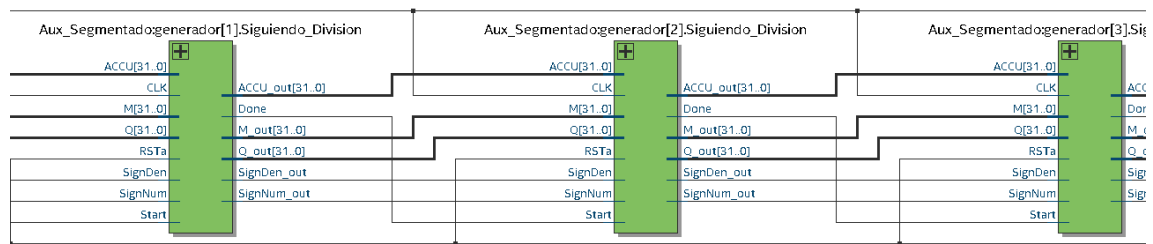
Para cerciorarnos de ello, adjuntamos la siguiente imagen →



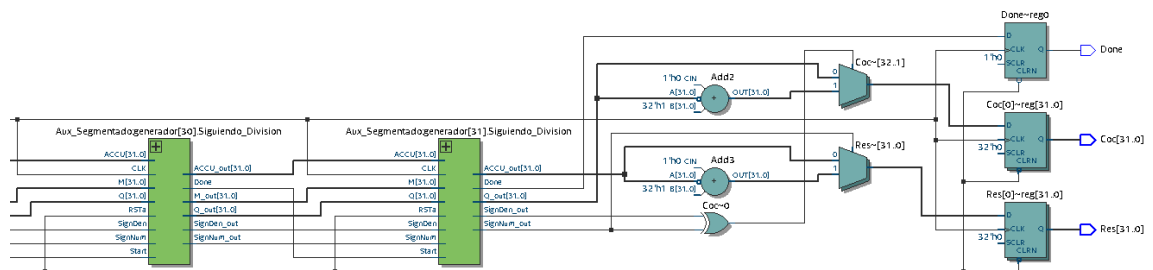
En cuanto al RTL viewer (sinterización del diseño), nos sale como pensamos la adición de fliflops en serie, en el que lo podemos observar en las siguientes tres imágenes →



En esta primera imagen observamos el comienzo del diseño y el inicio del primer FF.



Observamos en la segunda imagen la continuación de los flip flops.



Visualizamos finalmente como acaba con la obtención del resultado requerido.