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:

1. Diseño Componente Data-path
2. Diseño Componente Control-path
3. Descripción del sistema y verificación funcional

Diseño Verificación:

1. Simulación. Compilación del sistema y simulación lógica
2. Verificación lógica BÁSICA del diseño realizado
3. Verificación intermedia y avanzada

# **ETAPA RTL. DESCRIPCIÓN DEL SISTEMA COMPLETO Y VERIFICACIÓN FUNCIONAL.**

#### 1.1 Descripción del nivel de jerarquía top.

El diseño de nivel jerárquico top es el Divisor\_Algoritmico. El tamaño de Números de entrada es 32 bits.

Diagrama, Texto

Descripción generada automáticamente

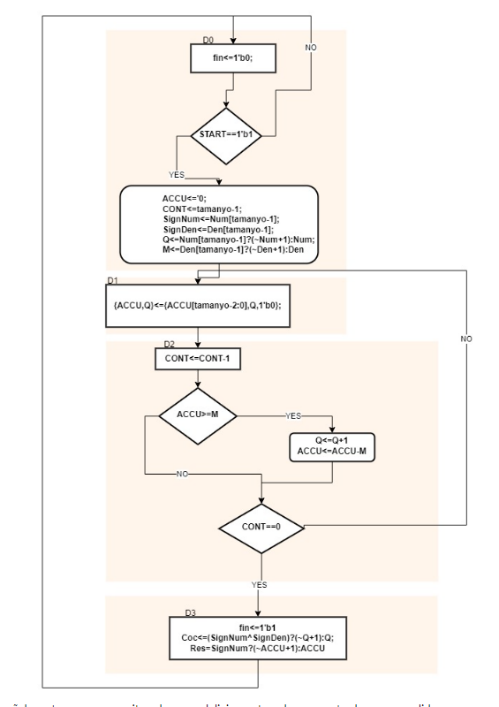
Fig.1 Bloque Divisor

Puertos de entrada y salida:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Fig.2 Puertos

El divisor de complementos que se va a implementar estará basado en el código ASM dado en la tarea. Se hace una implementación directa del ASM mediante un único fichero systemVerilog, que implementará de forma única tanto el Control-Path como el Data-Path.

Este ASM tendrá 4 estados.

**Estado 1** - Standby.

**Estado 2** - Actualización de los valores.

**Estado 3** - Operación de suma.

**Estado 4** - Fin.

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

Texto

Descripción generada automáticamente

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.

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

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

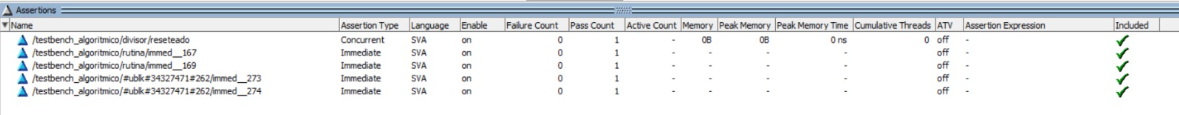
Además del covergroup encargado de las entradas, el equipo ha decidido implementar un covergroup con trigger que se encarga de almacenar los cocientes y restos que dan las operaciones que realizadas. La esperanza en la combinatoria de este no era tan alta como en las entradas ya que no dependen puramente 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.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

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 checkeo realizado es el siguiente 🡪



# **ETAPA LÓGICA. COMPILACIÓN DEL DIVISOR Y SIMULACIÓN LÓGICA.**

Interfaz de usuario gráfica

Descripción generada automáticamente

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.

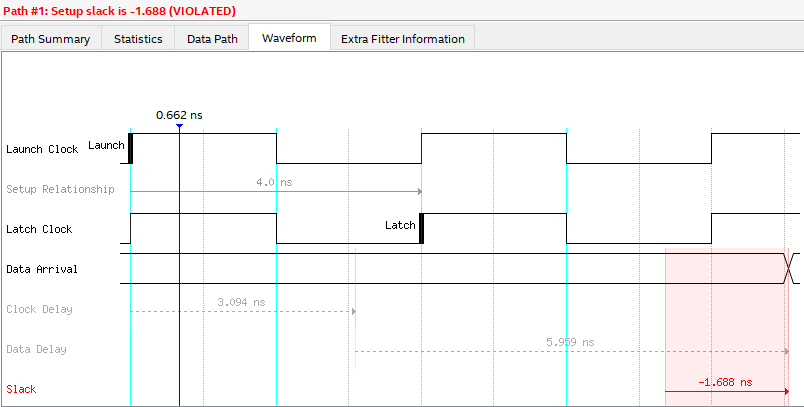
En primer lugar comprobaremos que relojes detecta. Podemos ver que detecta una señal de reloj denominada CLK con las siguientes características:



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.

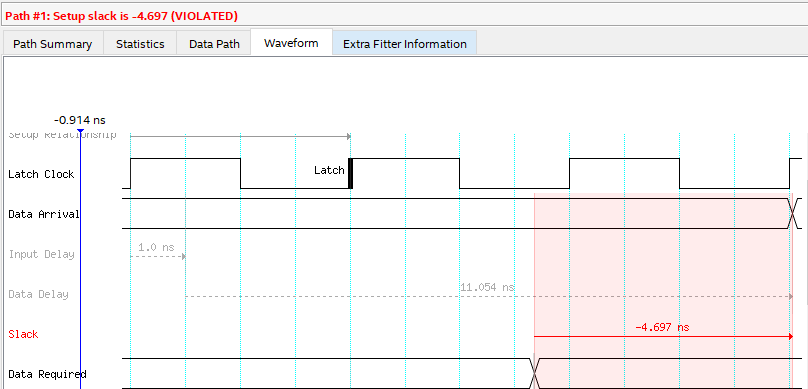


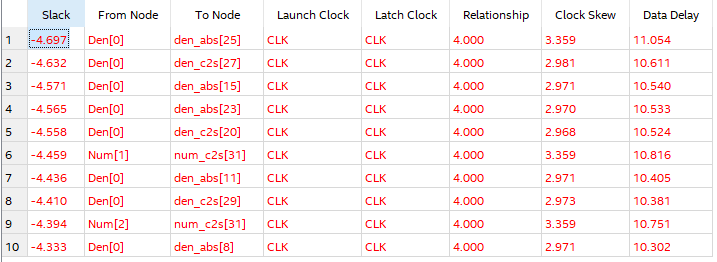


Si queremos introducir información sobre qué ocurre desde el exterior hasta los puertos de entrada del divisor algorítmico, tendremos más información de más paths.

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

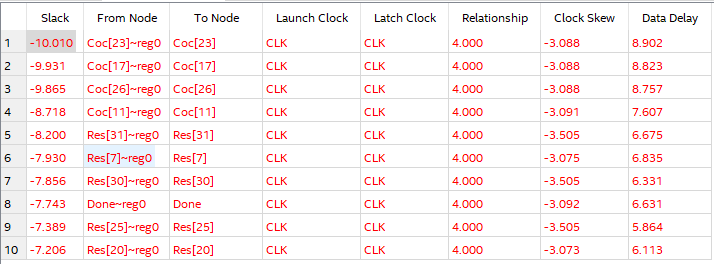


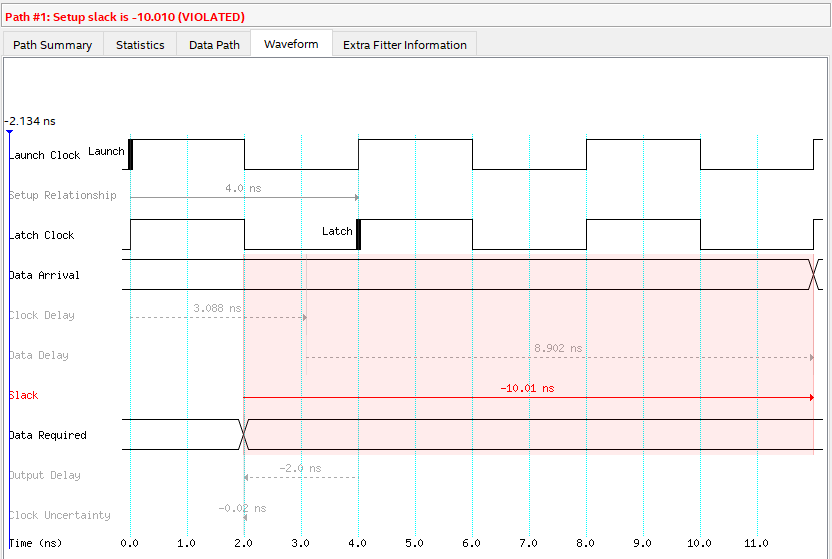




En Set Output Delay añadimos un retardo de 2 ns y listamos todos los puertos de salida (Coc, Res y Done).







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 Synthesys activado.

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



# **DISEÑO DEL DIVISOR SEGMENTADO**

#### 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:

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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 🡪

Texto

Descripción generada automáticamente

Para estado i=tamaño=etapas 🡪

Texto

Descripción generada automáticamente

Para estado ‘default’ el que incluye los estados restantes que falta entre medias de i=0 y i=max 🡪

Texto

Descripción generada automáticamente

#### 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 declara 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 🡪

Texto

Descripción generada automáticamente

Y el testbench principal sería de tal forma 🡪

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

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.

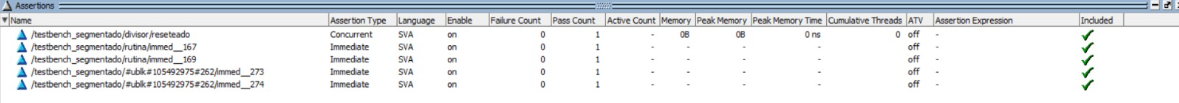
En cuanto al rango cubierto de coverage es el que mostramos a continuación 🡪

Tabla

Descripción generada automáticamente

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 checkeo realizado es el siguiente 🡪



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 🡪

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

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 🡪

Diagrama, Esquemático

Descripción generada automáticamente

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

Imagen que contiene Tabla

Descripción generada automáticamente

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

Diagrama

Descripción generada automáticamente

Visualizamos finalmente como acaba con la obtención del resultado requerido.