Proyecto 2 Microelectrónica

Leonardo Agüero Villagra
Escuela de Ingeniería Eléctrica
Universidad de Costa Rica
Carné: B70103

Gabriel Jiménez Amador
Escuela de Ingeniería Eléctrica
Universidad de Costa Rica
Carné: B73895

***Resumen*—En este proyecto se trabaja con el diseño RTL de un sistema que recibe un bloque de 12 bytes y un target y que mediante una función llamada *micro_ucr_hash* devuelve en su salida un nonce de 4 bytes que genera una salida de 3 bytes en *micro_ucr_hash* donde los primeros dos bytes son menores que el target especificado. Dos diseños del sistema son implementados, uno que optimiza el área y el otro la velocidad. Ambos RTL son verificados por medio de un testbench y al comprobar su funcionamiento son sintetizados mediante la herramienta *QFLOW*. Finalmente se reportan las métricas obtenidas de los procesos de síntesis, *placement* y *static timing analysis*.**

I. Introducción

El proyecto realizado corresponde a la continuación del trabajo del primer proyecto. En el primer proyecto se realiza un diseño de alto nivel del sistema optimizado para área y rendimiento. Este modelo permitió probar el funcionamiento lógico del sistema así como para verificar relaciones de entrada/salida del mismo. En esta segunda parte se implementará el RTL y la síntesis del sistema.

A continuación se muestran dos implementaciones de un RTL de un sistema que emplea cierta función hash llamada `micro_ucr_hash` para iterar sobre sus salidas para lograr generar salidas especiales según cierto bloque de entrada y *target* deseado. La vista general de bloques del sistema se observa en la Figura 1.

El sistema debe:

1. Recibir un bloque de 12 bytes, un target de 1 byte más una señal de inicio.
2. Concatenarle un nonce de 4 bytes.
3. Calcularle la firma de la concatenación del bloque + nonce, haciendo uso de la función hash `micro_ucr_hash`.
4. Si los dos primeros bytes de la salida:
   - No son menores al target: debe intentar de nuevo modificando el nonce y volviendo a concatenarlo con el bloque.
   - Son menores al target: el sistema debe retornar el nonce que hace cumplir la condición.
5. Al terminar, el sistema retorna una señal de terminación.

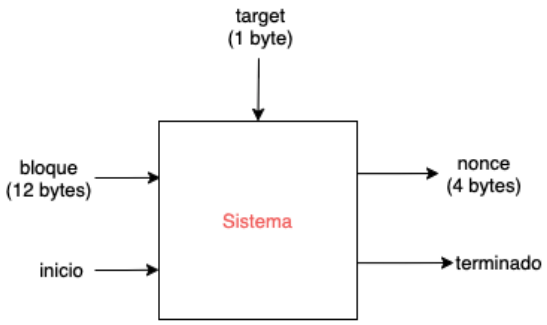

Figura 1. Sistema a implementar un RTL y posteriormente realizar un proceso de síntesis.

Una implementación del sistema fue diseñada priorizando un reducido uso de área (i.e., menores componentes y lógica simplificada), y otra priorizando desempeño aunque utilice mayor área y más componentes. Ambos diseños fueron posteriormente puestos en práctica, implementando un RTL en verilog y verificándolo mediante un testbench. Luego, estos sistemas fueron sintetizados mediante la herramienta *Qflow*, dentro de este programa se insertan los RTL realizados y se realizan las etapas de síntesis, *placement* y *static timing analysis*. Finalmente se obtienen distintas métricas de estas etapas como cantidad de celdas (combinacionales como secuenciales), cantidad de buffers e inversores, cantidad de cables, área del circuito, frecuencia máxima de operación y el tiempo de cálculo para 1000 ejecuciones. El repositorio trabajado con ambos modelos se puede explorar en https://github.com/leus8/micro-ucr-hash-rtl

II. Arquitectura optimizada en área

El principio del diseño de una arquitectura del Sistema presentado en la figura 1 consiste en lograr el funcionamiento correcto mediante la menor cantidad de bloques posibles, en este caso se debe evitar utilizar bloques redundantes para el funcionamiento del sistema ya que un bloque significa mayor área necesitada para construir el ASIC del sistema.

El diagrama de bloques del diseño de la arquitectura optimizada para el área se puede observar en figura 2.

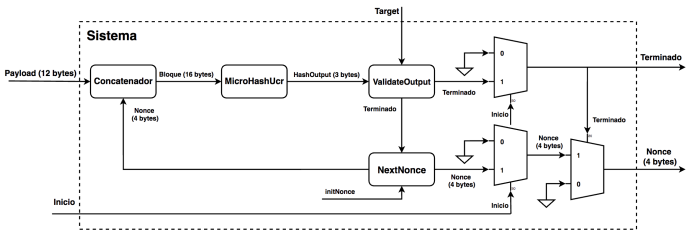

Figura 2. Arquitectura del sistema optimizada para área.

La arquitectura consiste en las dos entradas que debe tener el sistema que es el bloque de 12 bytes y la señal de inicio. En caso de que la señal de inicio sea 0 el circuito estará apagado y no se tendrá ninguna señal en la salida.

Al poner la señal de inicio en 1 entra los 12 bytes de la entrada (en esta implementación se le llamará *Payload*) donde entrará al bloque `Concatenador` que concatena el *payload* de 12 bytes de la entrada y el *nonce* respectivo para la iteración (para la primera iteración el nonce inicial será [0,0,0,0]) y producirá el bloque de 16 bytes que entra a la función `MicroHashUcr`.

La función `MicroHashUcr` producirá una salida de 3 bytes que en la implementación realizada se nombra como *HashOutput* que entra al bloque `ValidateOutput` el cual revisa que el valor de los dos primeros bytes sea menor que el *Target* que recibe en la entrada. Si `ValidateOutput` valida que la salida producida por `MicroHashUcr` es correcta se coloca la señal de *Terminado* en 1 y se envía a la salida el *nonce* con el que se ha validado el resultado. En caso de ser incorrecta la salida de `MicroHashUcr` se procede al siguiente *nonce* calculado por el bloque `NextNonce` y se vuelve a realizar el proceso hasta que se valide un *HashOutput* de `MicroHashUcr`.

III. Arquitectura optimizada en velocidad

Para la arquitectura enfocada en desempeño y velocidad en la generación de nonces válidos se tomó el diseñó preliminar de área y se decidió paralelizar ciertos componentes en lugar de realizar todo serialmente. Claro, esto significa mayor cantidad de módulos en su implementación en un circuito integrado, y por lo tanto mayor área del ASIC que habría que cubrir.

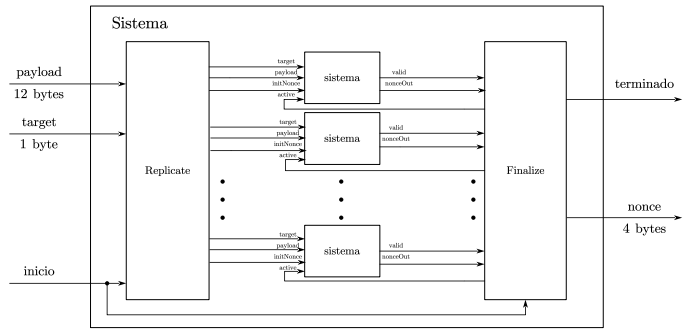

Figura 3. Arquitectura del sistema optimizada para desempeño.

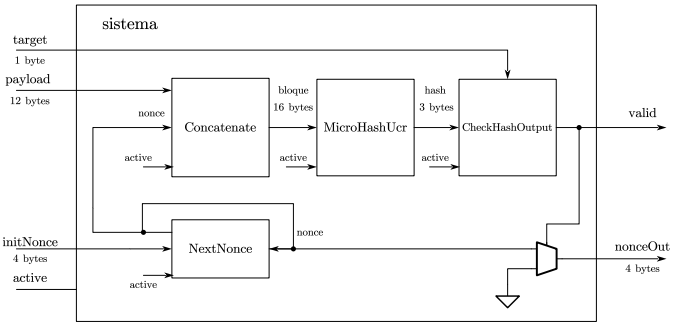

Figura 4. Submódulos paralelizados de Sistema.

El diagrama general de bloques propuesto para el sistema se observa en la Figura 3. En él se observan N submódulos `sistema` que correrán y procesarán nonces diferentes en paralelo. Estos submódulos son descritos por el diagrama de la Figura 4.

La funcionalidad de este diseño en teoría es la misma que la de la optimizada en área, salvo que logrará encontrar un nonce válido mucho más rápido gracias a la paralelización del proceso principal.

IV. Simulación y resultados

IV-A. Área

Una vez implementado el RTL en Verilog, se procede a crear un testbench que valide que los resultados son correctos. Desde el proyecto anterior se habían guardado unas entradas con sus respectivos `Nonce` generados. Como la simulación dependiendo del `payload` en la entrada puede tardar mucho, se decide empezar con un `initNonce = Nonce - 1000` de manera que la simulación no tarde demasiado. Por lo tanto se realiza la simulación con las siguientes entradas:

```
initNonce: 0x01001b26 - 0x03e8
Payload: [0x39 0x7d 0x9f 0x2f 0x40 0xca 0x9e 0x6c 0x6b
0x1f 0x33 0x24 0xfd]
Target: 10
Active: 1
```

Cabe destacar que `initNonce` no es una entrada definida por el usuario, sino más bien dentro del RTL se inicializa con ese valor. Las señales obtenidas con estas entradas se observan en la figura 5:

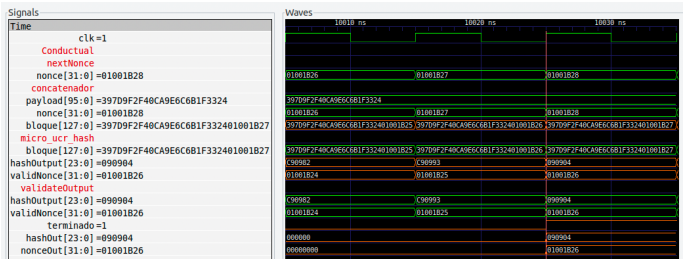

Figura 5. Señales obtenidas para RTL realizado.

En la figura 5 la señales color verde corresponde a entradas para el respectivo módulo y las color anaranjado corresponden

a salidas. Vemos que desde que se calcula el nonce correcto en el módulo *nextNonce* toma dos ciclos de reloj para aparecer a la salida de `validateOutput` que es la salida de todo el RTL (la simulación dura 10035 ns). Las salidas obtenidas de la figura 5 son:

```
nonce válido: [0x01 0x00 0x1b 0x26]
hash válido: [0x09 0x09 0x04]
```

Seguidamente se procede a realizar la síntesis del RTL con el programa *Qflow*. Al terminar el proceso de síntesis se comparan las señales del `.rtlnopower.v` con el RTL conductual. Los resultados se pueden observar en la misma figura 6:

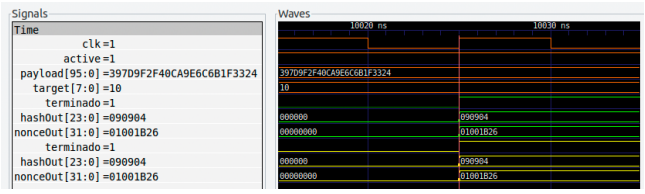

Figura 6. Simulación conductual y sintetizada del sistema enfocado en área.

De la figura 6 las señales color anaranjado corresponden a las entradas, las señales verdes son las salidas del modelo conductual y las señales amarillas son las salidas del modelo sintetizado. Se observa que los modelos conductual y sintetizado se comportan de la misma manera que es lo deseado.

Se procede a realizar las etapas de *placement* y *static timing analysis*, las métricas obtenidas se puede observar en el cuadro I. La información de la cantidad de celdas, cables, inversores y buffers se obtuvieron del archivo `synth.log`. Del archivo `sta.log` se obtiene la frecuencia máxima de operación. Para el cálculo del área se consulta el archivo `sistema_area.def` y se obtiene la siguiente información:

```
DESIGN sistema_area ;
UNITS DISTANCE MICRONS 100 ;

DIEAREA ( -320 -300 ) ( 47360 33300 ) ;
```

Lo cuál se puede calcular el área total del circuito en μm de la siguiente manera:

$$\frac{320 + 47360}{100} = 476{,}8\,\mu\text{m}$$
$$\frac{300 + 33300}{100} = 336\,\mu\text{m}$$

El tiempo de cálculo para 1000 ejecuciones se calcula de la siguiente manera:

$$\frac{2 \cdot 1000}{73{,}4454 \times 10^6} = 27{,}23\,\mu\text{s}$$

Del cálculo anterior se toma en base que son 1000 ejecuciones, el sistema dura 2 ciclos de reloj en validar un nonce y que la frecuencia máxima de operación es de 73,4454 MHz como se observa en el cuadro I.

Cuadro I
MÉTRICAS DEL SISTEMA OPTIMIZADO EN ÁREA OBTENIDAS DE LOS PROCESOS DE SÍNTESIS, PLACEMENT Y STATIC TIMING ANALYSIS MEDIANTE EL PROGRAMA QFLOW.

Métrica	**Resultado**
Cantidad total de celdas	4687
Cantidad de celdas combinacionales	270
Cantidad de celdas secuenciales	3670
Cantidad de buffers	57
Cantidad de inversores	690
Area total del circuito	476,8 μm x 336 μm
Cantidad total de cables	4111
Frecuencia máxima de operación	73,4454 MHz
Tiempo de cálculo para 1000 ejecuciones	27,23 µs

IV-B. Velocidad

Para el RTL de la arquitectura enfocada en desempeño, se modificó levemente el sistema diseñado en alto nivel del proyecto anterior. Siguiendo la Figura 3 el bloque de `Replicate` se implementó fácilmente como un bloque `generate` en Verilog y `Finalize` como un bloque `for` que revisa si las salidas son válidas.

Además, la generación de submódulos fue parametrizada para facilidad de simulación y síntesis. Nótese que si la cantidad de submódulos `sistema` es igual a 1, quedaría efectivamente igual que el sistema enfocado a área. Como se expresó en este inciso, **Icarus Verilog** tarda simulando una cantidad de tiempo muy grande, sin embargo para 2^N bloques con $N > 6$ se logró un tiempo de simulación relativamente pequeño para encontrar el Nonce válido.

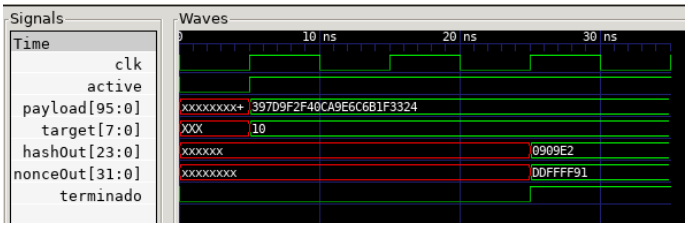

Figura 7. Simulación conductual del sistema enfocado en desempeño con 128 submódulos.

En la Figura 7 se tiene una simulación con un testbench igual al del sistema enfocado en área con 128 submódulos trabajando en paralelo. Con esta simulación se nota lo veloz que es este sistema en realidad, **sólo tarda 2 ciclos de reloj** (20 ns) en encontrar un nonce válido. Al sistema se le aplicaron las siguientes entradas:

```
Payload: [0x39 0x7d 0x9f 0x2f 0x40 0xca 0x9e 0x6c 0x6b
0x1f 0x33 0x24 0xfd]
Target: 10
```

Y se obtuvieron las siguientes salidas:

```
nonce válido: [0xdd 0xff 0xff 0x91]
hash válido: [0x09 0x09 0xe2]
```

Se observa que el nonce recibido en la sálida es válido puesto que el hash que forma tiene los dos primeros bytes menores al target (10).

Luego de sintetizar el sistema con 128 submódulos y aplicar las mismas entradas al sistema, se tienen las formas de onda

de la simulación de la Figura 8 en donde se observan las mismas salidas para el conductual (en verde) como para el sintetizado (en amarillo). Además a la izquierda se denotan los 128 submódulos.

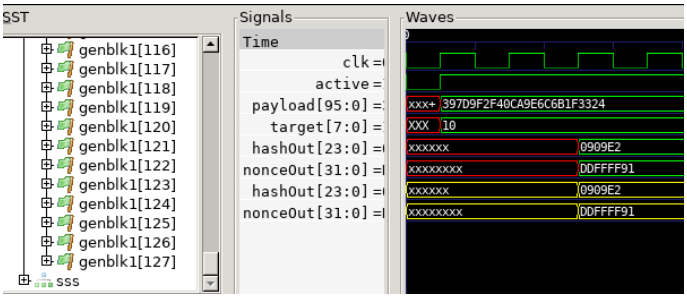

Figura 8. Simulación conductual y sintetizada del sistema enfocado en desempeño con 128 submódulos.

Tomando el archivo `synth.log` creado por la herramienta QFlow luego de la etapa de síntesis se obtienen las estadísticas de la síntesis con Yosys. De donde se arma el Cuadro II.

Cuadro II
MÉTRICAS DEL SISTEMA ENFOCADO EN DESEMPEÑO CON 128 SUBMÓDULOS

Métrica	Resultado
Cantidad total de celdas	610028
Cantidad de celdas combinacionales	578668
Cantidad de celdas secuenciales	31360
Cantidad de buffers	57
Cantidad de inversores	91353
Cantidad total de cables	538319

Aunque el sistema con 128 submódulos es simulado rápidamente ya que encuentra un nonce válido en poco tiempo, su síntesis tomó bastante tiempo y su placement y posterior cálculo del STA no fue posible obtenerlo en un lapso de 2 horas. Para ahorrar tiempo se decidió realizar las demás etapas del flujo con 4 submódulos. Esto lleva a menor esfuerzo para sintetizar y place & route. Las nuevas métricas se observan en el Cuadro III.

Cuadro III
MÉTRICAS DEL SISTEMA ENFOCADO EN DESEMPEÑO CON 4 SUBMÓDULOS

Métrica	Resultado
Cantidad total de celdas	19317
Cantidad de celdas combinacionales	18337
Cantidad de celdas secuenciales	980
Cantidad de buffers	57
Cantidad de inversores	2924
Cantidad total de cables	16924
Area total del circuito	834,4 μm x 796 μm
Cantidad total de cables	16924
Frecuencia máxima de operación	73.1825 MHz
Tiempo de cálculo para 1000 ejecuciones	27,33 µs (*)

En general existe una clara correlación entre la cantidad total de celdas y cantidad de submódulos instanciados, resumiendo esta métrica de los cuadros I, II y III y dividiendo los últimos dos entre la cantidad de submódulos que tienen: se tiene 4687, 4766, 4829 celdas respectivamente. Este comportamiento se debe a que el sistema enfoncado al desempeño en realidad es muy similar al de área sólo que paralelizado.

Para el cálculo de área del circuito se tomó la información del archivo Design Exchange Format (DEF) generado luego de la etapa de *placement* que da un `DIEAREA` rectangular como se muestra a continuación:

```
DESIGN sistema_speed ;
UNITS DISTANCE MICRONS 100 ;

DIEAREA ( -320 -300 ) ( 83120 79300 ) ;
```

De donde se calcula el área en micras (μm) como:

$$\frac{83120 + 320}{100} = 834{,}4\,\mu\text{m}$$
$$\frac{79300 + 300}{100} = 796\,\mu\text{m}$$

(*) Es importante denotar que aunque el tiempo de cálculo para 1000 ejecuciones de la función hash teóricamente es de 27.33 µs, esto se refiere a un sólo submódulo corriendo 1000 ejecuciones por sí mismo. Con 128 submódulos, por ejemplo, se tendría un tiempo de cálculo dado por:

$$2 \cdot \frac{1000}{73{,}1825 \times 10^6} \cdot \frac{1}{128} = 213{,}5\,\text{ns}$$

Puesto que, a como se diseñó, cada submódulo se divide esas 1000 ejecuciones en porciones equitativas.

V. VERIFICACIÓN FORMAL CON SYMBIYOSYS

V-A. *Prueba inicial de la herramienta*

Con la herramienta de verificación formal, SymbiYosys, se realizó un BMC (*Bounded Model Check*) al sistema enfocado a desempeño, para este fin se creó un nuevo testbench `sistema_formal_tb.v`.

Para probar la herramienta se verificó lo siguiente:

- Asuma un `payload` de: `[0x39 0x7d 0x9f 0x2f 0x40 0xca 0x9e 0x6c 0x6b 0x1f 0x33 0x24 0xfd]` y un target de `10`.
- Cuando el sistema levante la señal de finalización `terminado` se espera que el nonce válido sea igual a 0.

Esta prueba se espera que falle puesto que el sistema ante estas mismas entradas ya vimos que calcula un nonce válido de `[0xdd 0xff 0xff 0x91]`. Luego de correr SymbiYosys, se obtiene lo siguiente:

```
##   0:00:00  Solver: yices
##   0:00:01  Checking assumptions in step 0..
##   0:00:01  Checking assertions in step 0..
##   0:00:03  Checking assumptions in step 1..
##   0:00:03  Checking assertions in step 1..
##   0:00:05  Checking assumptions in step 2..
##   0:00:06  Checking assertions in step 2..
##   0:00:08  Checking assumptions in step 3..
##   0:00:08  Checking assertions in step 3..
##   0:00:10  Checking assumptions in step 4..
##   0:00:10  Checking assertions in step 4..
##   0:00:11  BMC failed!
##   0:00:11  Assert failed in test: sistema_formal_tb.v:25.29-26.34
##   0:00:11  Writing trace to VCD file: engine_0/trace.vcd
##   0:00:20  Writing trace to Verilog testbench: engine_0/trace_tb.v
##   0:00:20  Writing trace to constraints file: engine_0/trace.smtc
##   0:00:20  Status: failed
```

Al examinar el archivo VCD generado con el contraejemplo, con GTKWave se observa las formas de onda de la Figura 9

en donde se observa el nonce válido esperado (`DDFFFF91`) más otro generado (`DDFFFF92`) que también cumple con las entradas asumidas.

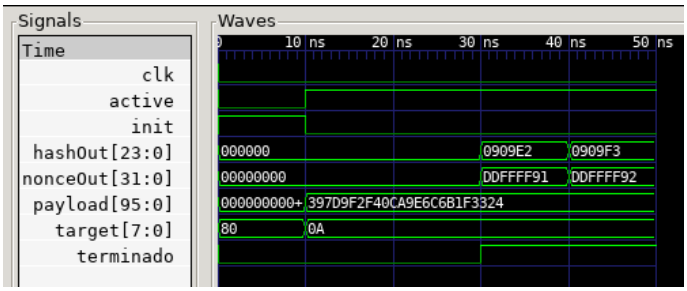

Figura 9. Contraejemplo dado por SymbiYosys para la prueba inicial.

V-B. Verificación formal

Ahora bien para verificar el sistema formalmente se debe limitar un poco las combinaciones de entradas. Por ejemplo, para targets muy pequeños, el sistema puede tardar mucho tiempo calculando nonces válidos. Se decidió verificar con los siguientes *constraints*:

- Todas las combinaciones de `payload` de 12 bytes.
- Targets entre 9 y 16 (target $\in [10:15]$).

La verificación confirmará que los hashes creados para cada nonce, son válidos según las entradas aplicadas. Es decir que los dos primeros bytes del hash sea menor al `target`. El testbench quedaría entonces como:

```
`include "sistema_speed.v"

// Testbench for formal verification
module test (
    input clk, active,
    input [7:0] target,
    input [95:0] payload,
    output reg terminado,
    output reg [23:0] hashOut,
    output reg [31:0] nonceOut
);

    //DUT instance, speed design is used with 2^7=128 submodules
    sistema_speed #(7) ss(clk, payload, active, target, terminado, nonceOut, hashOut);

    reg init=1;
    always @(posedge clk) begin
        // sistema_speed needs at least one clock cycle to load
        // initial nonces to every submodule
        if (init) assume (!active);
        else assume(active);
        init <= 0;
        // Once its active, assume a target range
        if (active) begin
            assume (target > 8'h09 && target < 8'h10);
        end
        // When finished, assert valid hash results
        if (terminado) begin
            case (target) // assert hash result for every target case
                10: assert(hashOut[23:16] < 10 && hashOut[15:8] < 10);
                11: assert(hashOut[23:16] < 11 && hashOut[15:8] < 11);
                12: assert(hashOut[23:16] < 12 && hashOut[15:8] < 12);
                13: assert(hashOut[23:16] < 13 && hashOut[15:8] < 13);
                14: assert(hashOut[23:16] < 14 && hashOut[15:8] < 14);
                15: assert(hashOut[23:16] < 15 && hashOut[15:8] < 15);
                default: assert(0); // target shoudn't be out of this case range
            endcase
        end
    end
endmodule
```

Corriendo un BMC con una profundidad de 10 ciclos con SymbiYosys con este testbench se obtiene lo siguiente:

```
##   0:00:00  Solver: yices
##   0:00:01  Checking assumptions in step 0..
##   0:00:01  Checking assertions in step 0..
##   0:00:03  Checking assumptions in step 1..
##   0:00:03  Checking assertions in step 1..
##   0:00:05  Checking assumptions in step 2..
##   0:00:05  Checking assertions in step 2..
##   0:00:07  Checking assumptions in step 3..
##   0:00:07  Checking assertions in step 3..
##   0:00:09  Checking assumptions in step 4..
##   0:00:09  Checking assertions in step 4..
##   0:00:13  Checking assumptions in step 5..
##   0:00:15  Checking assertions in step 5..
##   0:00:19  Checking assumptions in step 6..
##   0:00:24  Checking assertions in step 6..
##   0:00:37  Checking assumptions in step 7..
##   0:00:48  Checking assertions in step 7..
##   0:01:33  Checking assumptions in step 8..
##   0:01:45  Checking assertions in step 8..
##   0:02:27  Checking assumptions in step 9..
##   0:02:50  Checking assertions in step 9..
##   0:03:41  Status: passed
```

De donde se observa que la herramienta tardó unos 3:41 minutos en verificar el sistema con todas las combinaciones de entradas.

VI. Conclusiones

- Se logró implementar un RTL del sistema enfocado para área en el cual sacrifica velocidad y rendimiento debido a que utiliza la menor cantidad de módulos posibles de esta forma minimizando el área que necesita el ASIC.
- El sistema enfocado en área necesita de 2 ciclos de reloj para poder calcular y validar un nonce en la salida.
- Se logró implementar un sistema enfocado en desempeño RTL parametrizado que cálcula nonces empleando 2^N bloques o submódulos generados que trabajan paralelo, partiendo de nonces iniciales diferentes.
- Se logró trabajar con ambos diseños implementados del sistema hasta la etapa del cálculo del STA con la herramienta Qflow. Para el sistema enfocado en desempeño se tuvo que invertir más tiempo en esperar a que la herramienta lograra realizar la síntesis y el *placement*. Este tiempo de espera incrementaba exponencialmente por cada submódulo que se agregaba.
- Se puede notar de los cuadros I y III que el sistema optimizado en área necesitó una menor cantidad total de celdas, cables y área que el optimizado para desempeño que es lo esperado.
- Se denota la importancia que tiene cada archivo generado por el flujo de diseño para la extracción de métricas. Además de los archivos de *logging* generados por Qflow.
- Se comprobó la utilidad de una herramienta de verificación formal como SymbiYosys que ayuda a verificar la funcionalidad de circuitos digitales de forma automatizada utilizando técnicas como BMC (Bounded Model Check), al mismo tiempo que proveen contraejemplos de cuando los resultados no corresponden a lo esperado.