Universidad de Costa Rica | Facultad de Ingeniería | Escuela de Ingeniería Eléctrica

Bitácora de Laboratorio 2

Roberto Sánchez Cárdenas - B77059
Gabriel Jiménez Amador - B73895

San José, 17 de mayo de 2021
Laboratorio de Circuitos Digitales

Resumen

Este laboratorio presenta una serie de simulaciones por comportamiento y post-síntesis de descripciones de hardware escritas en Verilog, en ellas se utilizó código escrito en el lenguaje C para analizar la ejecución de multiplicadores basados en LUTs y basados en arreglos de multiplicación. Se comparó el tiempo de ejecución al emplear los módulos en comparación al multiplicador interno del core PicoRV32, al implementar un calculador de números factoriales y se concluyó que el diseñado a mano llega a ser hasta 4.4 veces más rápido.

Índice

1. Introducción

Este documento presenta los resultados de simular los diseños propuestos para multiplicar números de varias longitudes. Los multiplicadores se realizaron utilizando operaciones básicas de suma, las cuales son soportadas en Verilog y también por medio de Look up tables, metodología común en la implementación de circuitos en FPGAs. Los circuitos presentados son capaces de multiplicar números de 4 bits y 32 bits.

Además se diseñó firmware que utilizan las operaciones de multiplicación implementadas en el cpu PicoRV32 por medio del módulo RV32M. En otros casos, el firmware solo escribe o lee los números en la memoria.

2. Resultados y análisis

Esta sección presenta los resultados obtenidos para cada uno de los ejercicios propuestos para la práctica, además se realiza un breve análisis de los resultados obtenidos.

2.1. Ejercicio 1

El ejercicio 1 consiste en el crear un firmware que contenga operaciones de multiplicación, de tal manera que el CPU utilice instrucciones del módulo RV32M. La función factorial, donde se implementan las operaciones de multiplicación se ve de la siguiente manera:

```
00000000 <fact-0x10>:
   0: 00004137          lui sp,0x4
   4: 00010113          mv sp,sp
   8: 068000ef          jal ra,70 <main>
   c: 00100073          ebreak

00000010 <fact>:
  10: fd010113          addi sp,sp,-48 # 3fd0 <end+0x3e80>
  14: 02812623          sw s0,44(sp)
  18: 03010413          addi s0,sp,48
  1c: fca42e23          sw a0,-36(s0)
  20: 00100793          li a5,1
  24: fef42623          sw a5,-20(s0)
  28: 00100793          li a5,1
  2c: fef42423          sw a5,-24(s0)
  30: 0200006f          j 50 <fact+0x40>
  34: fe842783          lw a5,-24(s0)
  38: fec42703          lw a4,-20(s0)
  3c: 02f707b3          mul a5,a4,a5
  40: fef42623          sw a5,-20(s0)
  44: fe842783          lw a5,-24(s0)
  48: 00178793          addi a5,a5,1
  4c: fef42423          sw a5,-24(s0)
  50: fe842783          lw a5,-24(s0)
  54: fdc42703          lw a4,-36(s0)
  58: fcf77ee3          bgeu a4,a5,34 <fact+0x24>
  5c: fec42783          lw a5,-20(s0)
  60: 00078513          mv a0,a5
  64: 02c12403          lw s0,44(sp)
  68: 03010113          addi sp,sp,48
  6c: 00008067          ret
```

Al correr una simulación del CPU PicoRV32 con un firmware en el cual se calcula el factorial para los números 5, 7, 10 y 12, se obtienen los siguientes resultados.

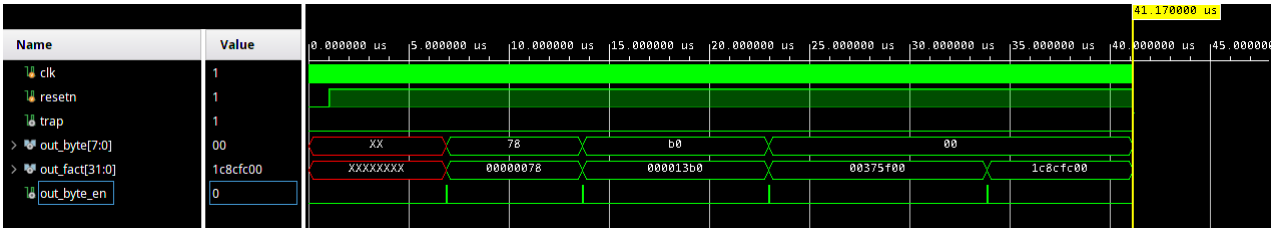

Figura 1: Simulación del cálculo de factorial

En la figura previa se puede observar que, cuando el registro solo tiene 8bits, solo es capaz de mostrar los últimos dos bytes del número. En el registro de 32bits, en número se calcula correctamente. Además, en el registro de 32 bits, tenemos los resultados del cálculo factorial, de modo que:

$$5! = 0x78 = 120$$
$$7! = 0x13b0 = 5040$$
$$10! = 0x375f00 = 3628800$$
$$12! = 0x1c82fc00 = 479001600$$

Si se observa detalladamente, entre mayor sea el número a calcular, toma más ciclos calcular el resultado. Para comprobar esto, se calcula el número de ciclos que toma para cada resultado.

Sabemos que cada ciclo toma $0{,}01\mu s$, por ende:

$$r1 = \frac{6{,}88 - 1}{0{,}01} = 588 \text{ ciclos}$$
$$r2 = \frac{13{,}67 - 6{,}88}{0{,}01} = 679 \text{ ciclos}$$
$$r3 = \frac{22{,}95 - 13{,}67}{0{,}01} = 928 \text{ ciclos}$$
$$r4 = \frac{33{,}89 - 22{,}95}{0{,}01} = 1094 \text{ ciclos}$$

Por lo tanto, el promedio de ciclos es de **823 ciclos** por resultado. Este promedio es solo válido para esta prueba, pues si se calculan números mayores, el promedio será mayor.

2.2. Ejercicio 2

En este ejercicio se realizó la síntesis e implementación del diseño utilizado en el ejercicio previo. En la siguiente figura se muestra la simulación funcional post síntesis. Además, se puede notar que el resultado concuerda con el mostrado en la simulación por comportamiento de la Figura 1.

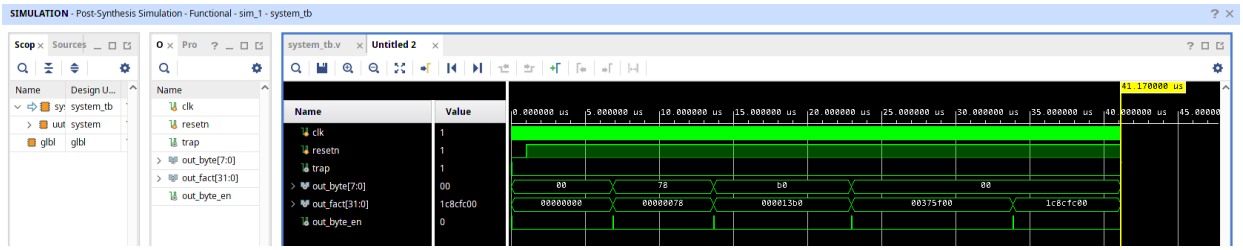

Figura 2: Resultado de la simulación post síntesis

Según el simulador, la implementación utiliza:

- Frecuencia: 100.000 MHz
- LUT: 1154
- Slices: 375
- FlipFlops: 828

2.3. Ejercicio 3

En el tercer ejercicio se creó un multiplicador de 4 bits con un LUT de 2 bits. Siguiendo el diseño propuesto en el anteproyecto, no se le hicieron modificaciones.

En la Figura 3 se tienen las entradas A y B de 4 bits multiplicadas en el resultado `result` de 8 bits. Se observan los resultados esperados de la operación.

Por ejemplo:

$$1 \cdot 2 = 2$$

$$6 \cdot 4 = 24$$

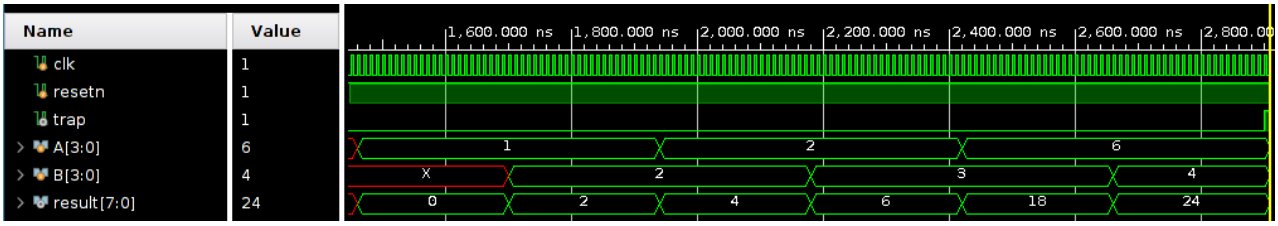

Figura 3: Simulación de la multiplicación de 4 bits con LUT

2.4. Ejercicio 4

En el tercer ejercicio se creó un multiplicador de 32 bits, creado a partir de submódulos multiplicadores de menor cantidad de bits (e.g., 16, 8, 4, 2) como se diseñó en el anteproyecto, no se hicieron modificaciones.

Respondiendo a la pregunta de la guía del laboratorio, teniendo una sola LUT monolítica para crear un multiplicador de 32 bits se ocuparían 2^{32} = 4294967296 filas y columnas para así obtener las multiplicaciones con todas las combinaciones de números de 32 bits. Por eso resulta más práctico trabajar con LUTs pequeñas utilizando submódulos para crear multiplicadores más grandes.

En la Figura 4 se tienen las entradas A y B de 32 bits multiplicadas en el resultado `result` de 64 bits. Se observan los resultados esperados de la operación.

Por ejemplo:

$$120 \cdot 2 = 240$$

$$39916800 \cdot 3628800 = 144850083840000$$

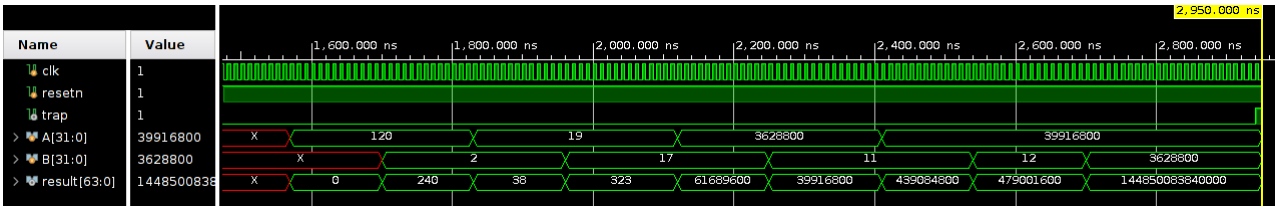

Figura 4: Simulación de la multiplicación de 32 bits con LUT

2.5. Ejercicio 5

En este ejercicio se realizó la implementación en Verilog de un circuito multiplicador de 4 bits a base de sumadores. El diagrama del circuito implementado se puede observan en la siguiente figura.

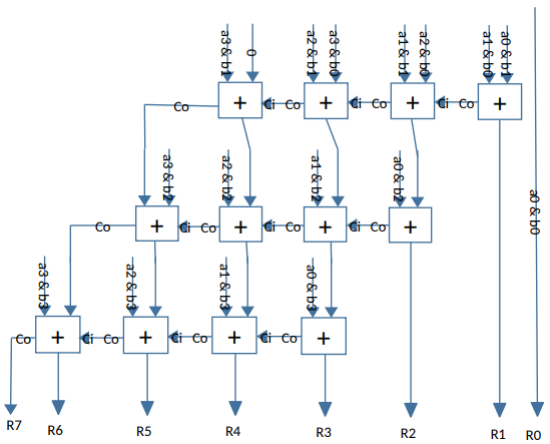

Figura 5: Diagrama de circuito multiplicador de 4 bits

Además se escribió un firmware en C para el manejo de los datos que se desean multiplicar. El firmware se puede leer en el siguiente código:

```
#include <stdio.h>
#include <stdint.h>

#define A_DIR 0x0FFFFFF0;
#define B_DIR 0x0FFFFFF4;
#define OUTBYTE_DIR 0x10000000;
void main()
{
    uint32_t *ptr1 = (uint32_t *)A_DIR;
    uint32_t *ptr2 = (uint32_t *)B_DIR;
    uint32_t *outbyte_ptr = (uint32_t *)OUTBYTE_DIR;

    for(int i = 0; i<=15; i++){
        *ptr1 = i;
        for(int j = 0; j<=15; j++){
            *ptr2 = j;
            *outbyte_ptr = *((volatile uint32_t *)0x0FFFFFF8);
        }
    }
}
```

Al simular el circuito implementado junto con el firmware previo, se obtuvieron los siguientes resultados. En la figura 6 se puede observar el funcionamiento adecuado del reset y las primeras iteraciones de A y B. En la figura 7, se observa las iteraciones de A en 5 y 6 con sus respectivos resultados, los cuales son correctos. Además se puede notar que la salida del resultado se copia en la señal out_byte adecuadamente.

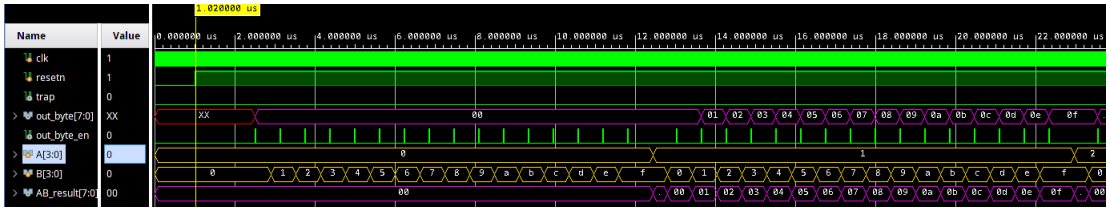

Figura 6: Simulación de array multiplier con señal de reset en bajo

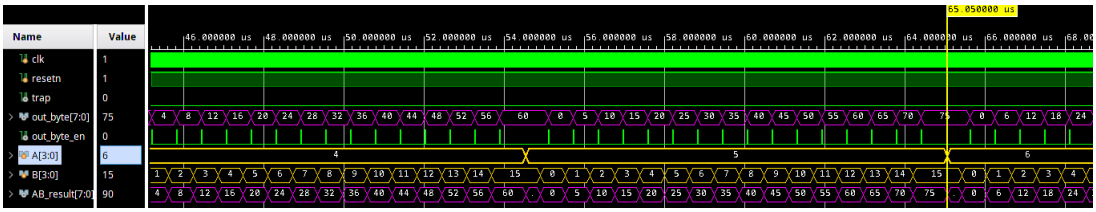

Figura 7: Funcionamiento del array multiplier en A = 4 y A = 5

Además de las pruebas previas, se comprobó que el diseño corriese exitosamente en la síntesis e implementación. En las siguiente figura se muestran los resultados de correr una simulación funcional post síntesis.

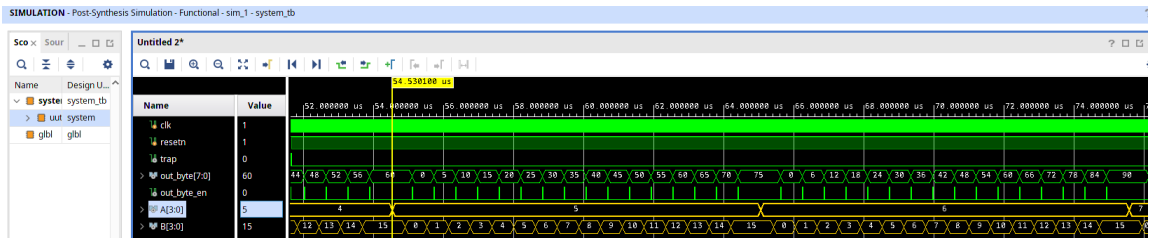

Figura 8: Resultados de correr una simulación post síntesis del array multiplier

Finalmente, se tiene la información respecto a las LUT, FF y slices utilizados.

- Frecuencia: 100.000 MHz
- LUT: 3612
- Slices: 1036
- FlipFlops: 859

2.6. Ejercicio 6

El ejercicio 6 tiene como fin realizar la implementación de un circuito multiplicador, por medio de la técnica array multiplier. Para ello se propone el uso de la funcionalidad de Verilog: generate, la cual permite instanciar múltiples veces un módulo y conectarlo.

El generate se suele trabajar por medio de iteraciones, por lo que los circuitos se pueden pensar como arrays de instancias o matrices de 2 o 3 dimensiones.

```
module adder_wCarry( input bit1, input bit2, input carryIn, input reset, output reg result, output reg carryOut );
    always @(*) begin
        if(reset) begin
            {carryOut, result} = bit1 + bit2 + carryIn;
        end
        else begin
            carryOut = 0;
            result = 0;
        end
    end
endmodule

module array_multiplier_32b(
    input [31:0] A,
    input [31:0] B,
    input reset,
    output [63:0] result);

    wire [31:0] adder_results[31:0];
    wire [31:0] adder_input1[31:0];
    wire [31:0] adder_input2[31:0];
    wire [31:0] adder_Co[31:0];
    wire [31:0] adder_Ci[31:0];

    genvar CurrentRow, CurrentCol;
    generate
        //Cada bit de resultado se obtiene en la columna 0. Esta columna no tiene Ci
        for(CurrentRow = 0; CurrentRow < 32; CurrentRow=CurrentRow+1) begin: row

            assign adder_Ci[CurrentRow][0] = 0;

            for(CurrentCol = 0; CurrentCol < 32; CurrentCol=CurrentCol+1) begin: column
                adder_wCarry adder(
                    .reset (reset),
                    .bit1 (adder_input1[CurrentRow][CurrentCol]),
                    .bit2 (adder_input2[CurrentRow][CurrentCol]),
                    .carryIn (adder_Ci[CurrentRow][CurrentCol]),
                    .carryOut (adder_Co[CurrentRow][CurrentCol]),
                    .result (adder_results[CurrentRow][CurrentCol])
                );

                assign adder_input1[CurrentRow][CurrentCol] = A[CurrentCol]&B[CurrentRow];
                assign adder_input2[0][CurrentCol] = 0;


                if(CurrentCol != 0) begin
                    assign adder_Ci[CurrentRow][CurrentCol] = adder_Co[CurrentRow][CurrentCol-1];
                end

                if(CurrentRow != 0) begin
                    if (CurrentCol!=31)begin
            assign adder_input2[CurrentRow][CurrentCol]=adder_results[CurrentRow-1][CurrentCol+1];
                    end
                    else begin
            assign adder_input2[CurrentRow][CurrentCol]=adder_Co[CurrentRow-1][CurrentCol];
                    end
        end

                assign result[CurrentRow] = adder_results[CurrentRow][0];

                if(CurrentRow==31) begin
                    assign result[CurrentCol+31] = adder_results[31][CurrentCol];
                end

            end
        end
    endgenerate
    assign result[63] = adder_Co[31][31];
endmodule
```

En el código anterior se tienen dos módulos, un sumador con carry y el multiplicador, el multiplicador instancia el módulo sumador en una matriz de tamaño 32x32, donde se realiza un manejo de cables estratégico de modo que se de el funcionamiento correcto.

Para comprobar el correcto funcionamiento del diseño, se propuso una serie de números para multiplicar, los cuales son ingresados por medio del firmware. Cabe destacar que la cuenta se realiza a nivel de firmware y para poder realizar un manejo adecuado en las simulaciones post síntesis se separó el resultado en dos grupos de 32 bits. En la siguiente figura se muestra el resultado de la simulación por comportamiento.

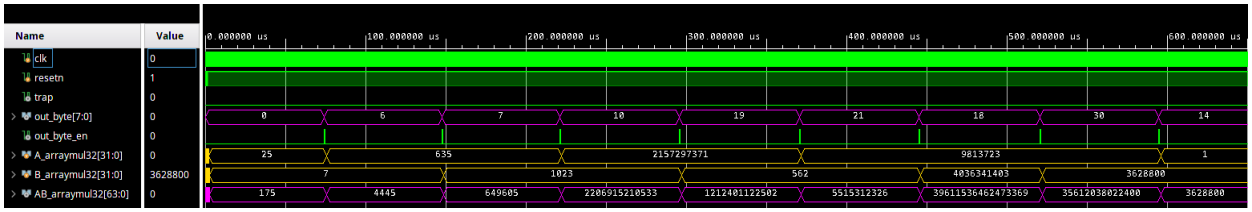

Figura 9: Simulación por comportamiento con resultados de la multiplicación a 64 bits

En la figura 9 se pueden observar los resultados claros de la multiplicación en el registro de 32 bits AB_arraymul. En la figura 10, el registro out_byte posee la cuenta de 1s en los resultados de la multiplicación en decimal, además se observa la separación que se realizó con el resultado para un manejo adecuado en la simulación post síntesis.

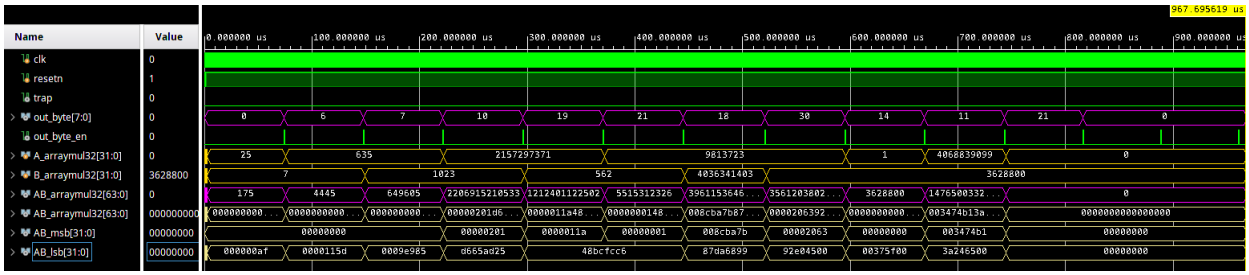

Figura 10: Simulación por comportamiento completa

Finalmente, en la figura 11 se tiene la simulación post síntesis. En esta no se puede observar el resultado a 64 bits por las limitaciones del CPU, pero sí se tienen los resultados del número de bits en 1 del resultado de la multiplicación. Las simulaciones concuerdan, por lo que se puede decir que el diseño se sintetizó adecuadamente.

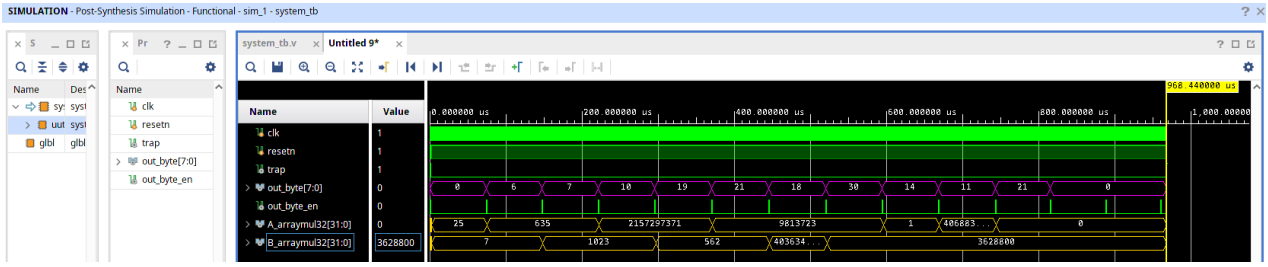

Figura 11: Simulación post síntesis

- Frecuencia: 100.000MHz

- LUT: 5631
- Slices: 1527
- FlipFlops: 892

¿Son los bloques generate sintetizables?

Como se observó previamente, los bloques generate sí son sintetizables. Lo que hace el generate es tomar un módulo y copiarlo varias veces en un circuito.

¿Cuántas etapas tiene este nuevo circuito de multiplicación? ¿Qué podría ocurrir con el periodo del reloj si se añaden más y más etapas de lógica combinatoria?

Este circuito posee 32 etapas. Si se añaden más etapas, por los tiempos de conmutación propios de las compuertas, se pueden generar errores de temporización a ciertas frecuencias, por lo que el circuito debería ser más lento. Se pueden añadir etapas de lógica secuencial en medio para solventar.

¿Qué podría ocurrir con la frecuencia del circuito si se añaden latches entre cada etapa de sumadores?

Había más lógica en medio, por lo que sería más lento. Lo ideal sería añadir Flip-Flops para evitar errores de temporización, sin embargo estos haría en circuito mucho más lento, pues cada etapa tomaría un ciclo completo.

2.7. Ejercicio 7

En el Ejercicio 7 se implementó un calculador de factoriales de 32 bits en hardware y se probaron los mismos números que en el Ejercicio 1, de igual forma, no hubieron modificaciones a la solución propuesta en el anteproyecto. En la Figura 12 se ve como con una entrada en `fuente` de 5 se obtiene una salida en `result` de 120 ($5! = 120$). Se observan los cambios de estado en el calculador. Además el resultado se obtiene cuando pasan 700 ns luego de ingresado el número deseado, es decir 70 ciclos de reloj, considerando que el reloj tiene un periodo de 10 ns.

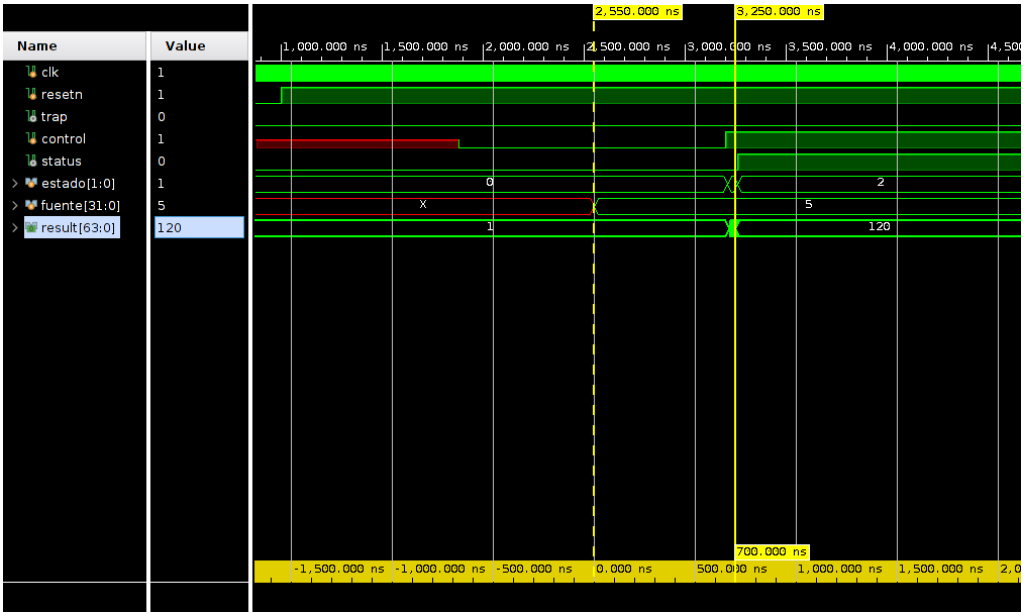

Figura 12: Simulación del calculador de factoriales , calculando el valor de 5!.

En la Figura 13 se tiene el cálculo de los números 5, 7, 10 y 12 de donde se obtuvo que [1]:

$$5! = 120$$
$$7! = 5040$$
$$10! = 3628800$$
$$12! = 479001600$$

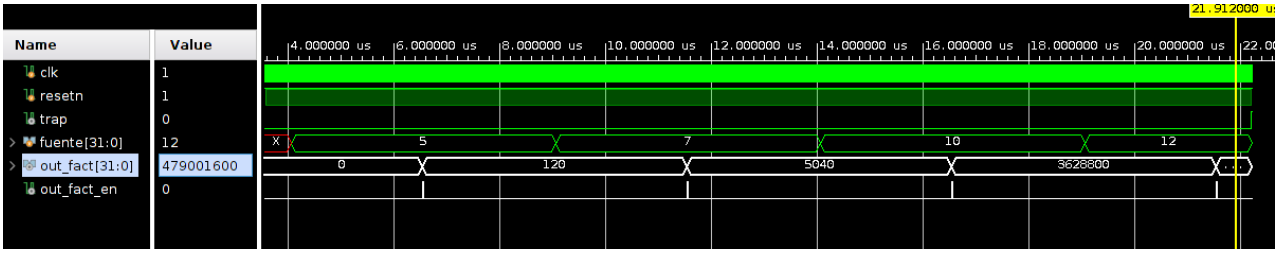

Figura 13: Simulación del calculador de factoriales, calculando 5, 7, 10 y 12.

Calculando los retardos, tanto en lo que le toma a la calculadora de factoriales y lo que le toma al CPU más la obtención del valor en `out_fact` con el módulo de memoria de `system.v` se obtiene los resultados de la Tabla 1 en donde se ve que el calculador incrementa (aunque poco) el tiempo que tarda en obtener el resultado a medida que incrementa el tamaño el número.

1 El valor de 12! se observa en el espacio de la izquierda al estar resaltado este valor en las formas de onda.

Factoriales	Ciclos de reloj	
	Calculador	UUT
5!	70	247
7!	72	247
10!	75	247
12!	77	247
Promedio	**73,5**	**247**

Tabla 1: Ciclos de reloj empleados para calcular diferentes factoriales.

Sin embargo, al final y al cabo esto no afecta la cantidad de ciclos que se tardan en obtener el dato de memoria combinando todo el UUT (Todos los módulos de verilog + core + módulo de memoria) que se mantiene igual en 247 ciclos. Comparando los retardos con los del Ejercicio 1, se tiene un periodo de ejecución mucho más corto, y además constante.

3. Repositorio

El código necesario para emular los resultados de este laboratorio se pueden encontrar en Github utilizando el siguiente enlace:
https://github.com/ucr-ie-0424/laboratorio-2-jimenez-sanchez.git
El ID del último commit es:
809de50392afd5c55eeefc275d25f481d547c7cb

4. Conclusiones y recomendaciones

4.1. Conclusiones

- Implementar un módulo especializado en hardware provee resultados en un periodo mucho menor que si se empleara sólo el core. El cálculo de factoriales resultó hasta 4.4 veces más rápido que su ejecución con el PicoRV32.
- El número de ciclos necesarios para obtener resultado final del UUT puede ser constante, si el cálculo principal en hardware da resultados muy rápidos. El tiempo de ejecución dependerá más de la lectura de memoria que el propio cálculo pedido.
- El uso de `generate` permite crear múltiples instancias de un módulo con pocas líneas de código. El resultado es idéntico al de instanciarlos manualmente.
- A nivel de resultados, un `generate` debe poder generar el mismo resultado que una instanciación a mano. La ventaja es la reducción de líneas de código.

4.2. Recomendaciones

- Al trabajar con `generate`, es conveniente pensar los diseños de forma matricial y hallar un patrón que permita una interconexión adecuada. Si en algún punto un patrón no se cumple, se puede añadir condicionales que modifique la creación del bloque.
- Resulta útil modelar calculadoras complejas como máquinas de estado, de esta forma se conoce en que paso del cálculo se encuentra y resulta más ordenado su implementación en código.