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

Anteproyecto II

Roberto Sánchez Cárdenas — B77059
Gabriel Jiménez Amador — B73895

San José, 31 de agosto de 2021
IE0424 — Laboratorio de Circuitos Digitales

Índice

1. Introducción

Este laboratorio tiene como fin realizar una introducción al diseño e implementación de circuitos combinatorios a nivel de simulación en la herramienta de Xilinx: Vivado.

2. Objetivos

Usar Vivado para el desarrollo de circuitos combinatorios.

- Utilizar las herramientas del Xilinx Vivado.
- Refrescar los conocimientos relacionados al diseño de lógica digital.
- Familiarizar a los estudiantes con las interacciones de software y hardware.

3. Anteproyecto

3.1. Investigue qué instrucciones forman parte de la extensión RV32M y describa cada una de estas instrucciones brevemente.

Como fue mencionado en el laboratorio 1, RISC-V es una arquitectura modular que permite ser extendida con nuevas instrucciones conforme sea necesario. La extensión RV32M es un conjunto de instrucciones que permiten realizar multiplicaciones y divisiones a 32 bits y como se menciona en [9], también existe un ser a 64 bits llamado RV64M, que realiza las mismas operaciones. Las instrucciones disponibles son:

- mul: Esta operación realiza una multiplicación de dos registros de x-longitud y los coloca en la parte baja de otro registro de x-longitud de bits.
- mulh: Realiza la misma operación que mul, pero retorna la parte alta del resultado de la multiplicación. Es para multiplicación de 2 números con signo.
- mulhsu: Realiza la misma operación que mul, pero retorna la parte alta del resultado de la multiplicación. Es para multiplicación deun número con signo y otro sin signo.
- mulhu: Realiza la misma operación que mul, pero retorna la parte alta del resultado de la multiplicación. Es para multiplicación de 2 números sin signo.
- div: División de dos números de x-longitud de bits con signo.
- divu: División de dos números de x-longitud de bits sin signo.
- rem: Remanente de una división de dos númeors con signo.
- remu: Remanente de una división de dos números sin signo.

3.2. Describa ¿cómo se definen y cómo se usan los parámetros de instancias en módulos implementados en el lenguaje Verilog.

El uso de parámetros en Verilog es muy útil pues permite asignar una constante a una palabra clave, de modo que esta pueda ser llamada a lo largo del código. Para mostrar como se implementan, se presenta el siguiente código simple

```
module counter#(parameter N=2, parameter output_size=8)
            (input clk,
            input reset,
            output reg [output_size-1:0] out);

    always @(posedge clk) begin
        if(reset) begin
            out <= 0;
        end
        else begin
            out <= out + N;
        end
endmodule
```

En el ejemplo previo se utiliza el parámetro para incrementar el valor de la salida, mientras que el parámetro output_size se usa para indicar el número de bits que posee la salida.

3.3. Investigue qué es una LUT (look up table).

Las Look Up Table son una metodología útil para la evaluación de funciones, pues el principio de funcionamiento radica en el hecho de que se tiene una tabla con valores precalculados, de modo que al realizar una operación, se tiene un resultado ya previamente calculado que puede ser utilizado por el código. Actualmente es una metodología popular pues permite optimizar el análisis de señales. Principalmente se precalculan operaciones complejas que le serán recurrentes al proceso. Un caso de uso es el procesamiento de imágenes. [2]

En el caso de las FPGAs, las LUT son un componente muy importante que facilita la implementación de circuitos lógicos. Una LUT de m entradas se traduce a un array de 2^m, en donde se programa la tabla de verdad de una función lógica, de modo que el mapeo a una resultado es muy eficiente. [7]

3.4. Investigue qué es el alineamiento de memoria y para qué sirve.

El alineamiento de memoria corresponde a la forma de acomodar datos en memoria. Con el concepto de que un bloque de memoria es continuo, cada subsección es accesible mediante una dirección que describe un tamaño fijado de bits. Si la memoria no está alineada y se quiere

accesar a memoria, deberá realizar más instrucciones y deberá leer más bloques de memoria de lo que hubiera hecho con una memoria alineada. [**ibm**] Para ejemplificar, considere un CPU que lee al tamaño de un *word* (4 bytes en un procesador de 32 bits), y se desea realizar una lectura de un **char**, **int** y un **short** en memoria.

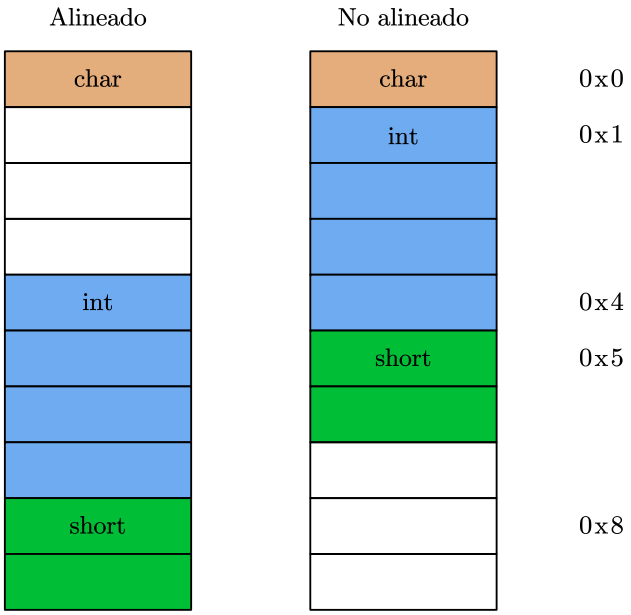

Figura 1: Comparación de memoria alineada y no alineada

En la figura 1 la memoria alineada de la izquierda resultará más sencillo y más eficiente accesar a los datos puesto que están separados a 4 *words* de distancia de cada uno. El CPU lo leerá en una sola instrucción. En contraste, el bloque de memoria de la derecha no está alineado, para que el CPU acceda al entero, tendría que primero acceder a 0x0, y luego pasar a 0x1 incrementando el número de instrucciones y accesos a realizar.

3.5. Conceptos importantes

- **Síntesis de lógica**

  Es el proceso de tomar una descripción de texto escrito en algún lenguaje de descripción de hardware y convertirlo a una estructura de lógica de registros de transferencia (rtl). [4]

- **RTL (register transfer level)**

  RTL es un tipo de diseño ampliamente utilizado para el diseño de procesadores debido a la complejidad que estos tienen y las facilidades que RTL tiene. Este es conocido como diseño de alto nivel, pes existen tipos de diseño de más bajo nivel como transistor-level design y el logic-level design, que permiten la creación de bloque sencillos. Por su parte,

RTL permite tener bloques complejos capaces de mover dados de un registro a otro por medio de los paths. [8]

- **Place and route**

  El proceso de *place and route* pone cada macro del netlist de síntesis in un lugar disponible en el chip de silicio determinado. Cada macro es conectado utilizando recursos de *routing*. En esta etapa de implementación del netlist al FPGA se pasa por 4 pasos importantes [6]:

  1. Se lee el netlist
  2. Extrae los componentes del netlist
  3. Pone los componentes en el dispositivo
  4. Interconecta los componentes según especificado

  Luego de estos pasos, el *place and route* es probado según los *timing constraints* definidos previamente. Si no los cumple al nivel esperado, se debe reiniciar e intentar un nuevo *placement*.

- **Floorplan**

  Algunos tools de *place and route* permite al usuario definir donde quieren tener componentes grandes manualmente, este proceso se llama *floorplanning*. Este proceso permite elegir los lugares de bloques tal que la distancia entre componentes sea lo más corto posible. Luego de que el usuario de el *floorplan* el algoritmo de *place and route* termina de ubicarlos y de establecer las conexiones deseadas [6].

- **FPGA**

  Las FPGAs son un tipo de circuitos integrados programables, es decir, por medio de código escrito en algún lenguaje de descripción de hardware se puede configurar el chip para implementar una función lógica. Los componentes más comunes dentro de una FPGA son las LUT y matrices de cambio, esto pues la idea básica de una FPGA es ser una memoria que implementa lógica combinacional. [8]

- **Slices de FPGAs**

  Los recursos de hardware de un FPGA son definidos por el número de *slices* que el dispositivo contiene, estos slices se conectan de forma que se obtienen bloques funcionales. Cada slice está compuesto por LUTs, flip-flops, multiplexores así como un lógica aritmética y de *carry*.

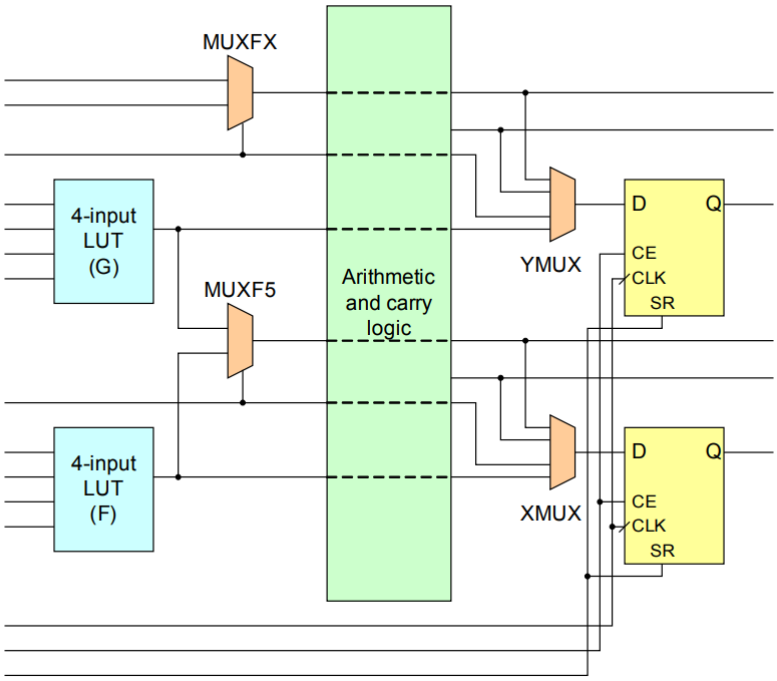

Figura 2: Diagrama simplificado de un slice del Xilinx Virtex-4 FPGA. [3]

3.6. Investigue cuál es la FPGA que tiene la tarjeta Nexys 4 DDR

Según la documentación de la tarjeta Nexys 4, esta posee un chip FPGA conocido como Artix 7, desarrollado por Xilinx que además posee el número de parte **XC7A100T-1CSG324C**.

Esta tarjeta posee 15,850 slices lógicos, cada uno con LUTs de 6 entradas y 8 flip-flops además posee 240 slices DSP (Digital Signal Processing). [5]

Con respecto al número máximo de pines de entrada y salida (I/O), se refirió al documento *7 Series FPGAs Data Sheet: Overview* [1] que muestra que la familia Artix-7 posee 500 pines de I/O, pero viendo la tabla de la Figura 3 y se busca para el número de parte descrito, se obtiene el número resaltado de 210 pines efectivos que se pueden usar como I/Os y no son partes de la interfaz de periféricos como USB o Ethernet, por ejemplo.

Table 5: **Artix-7 FPGA Device-Package Combinations and Maximum I/Os**

Package[1]	CPG236		CPG238		CSG324		CSG325		FTG256		SBG484		FGG484[2]		FBG484[2]		FGG676[3]		FBG676[3]		FFG1156	
Size (mm)	10 x 10		10 x 10		15 x 15		15 x 15		17 x 17		19 x 19		23 x 23		23 x 23		27 x 27		27 x 27		35 x 35	
Ball Pitch (mm)	0.5		0.5		0.8		0.8		1.0		0.8		1.0		1.0		1.0		1.0		1.0	
Device	GTP (4)	I/O HR(5)	GTP (4)	I/O HR(5)	GTP (4)	I/O HR(5)	GTP (4)	I/O HR(5)	GTP (4)	I/O HR(5)	GTP	I/O HR(5)	GTP (4)	I/O HR(5)	GTP	I/O HR(5)	GTP (4)	I/O HR(5)	GTP	I/O HR(5)	GTP	I/O HR(5)
XC7A12T			2	112			2	150														
XC7A15T	2	106			0	210	4	150	0	170			4	250								
XC7A25T			2	112			4	150														
XC7A35T	2	106			0	210	4	150	0	170			4	250								
XC7A50T	2	106			0	210	4	150	0	170			4	250								
XC7A75T					0	210			0	170			4	285			8	300				
XC7A100T					0	210			0	170			4	285			8	300				
XC7A200T											4	285			4	285			8	400	16	500

Notes:
1. All packages listed are Pb-free (SBG, FBG, FFG with exemption 15). Some packages are available in Pb option.
2. Devices in FGG484 and FBG484 are footprint compatible.
3. Devices in FGG676 and FBG676 are footprint compatible.
4. GTP transceivers in CP, CS, FT, and FG packages support data rates up to 6.25 Gb/s.
5. HR = High-range I/O with support for I/O voltage from 1.2V to 3.3V.

Figura 3: Combinación de FPGAs Artix-7 con el número máximo de I/Os según el dispositivo y empaquetado.[1]

4. Diseño

4.1. Ejercicio 1 y 2

Para esta sección se creó un código en C que calcula el número factorial por medio de iteración. Para ello se utilizan el tipo uint32_t para asignar variables.

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

#define MEMORY_ADD 0x10000000

uint32_t fact(uint32_t number){
    uint32_t factorial_result = 1;

    for(int counter = 0; counter != number; counter++)
   factorial_result *= counter;

    return factorial_result;
}

void main(){
```

```
16      uint32_t z = 5;
17      uint32_t x = 7;
18      uint32_t y = 10;
19      uint32_t h = 12;
20
21      uint32_t* ptr = (uint32_t*)MEMORY_ADD; //type cast the addr to a
        uint pointer
22
23      uint32_t num1 = fact(z);
24      *ptr = num1;
25
26      uint32_t num2 = fact(x);
27      *ptr = num2;
28
29      uint32_t num3 = fact(y);
30      *ptr = num3;
31
32      uint32_t num4 = fact(h);
33      *ptr = num4;
34  }
```

4.2. Ejercicio 3

Para construir un multiplicador de 4 bits utilizando mux de 4 entradas, se puede utilizar el siguiente diseño

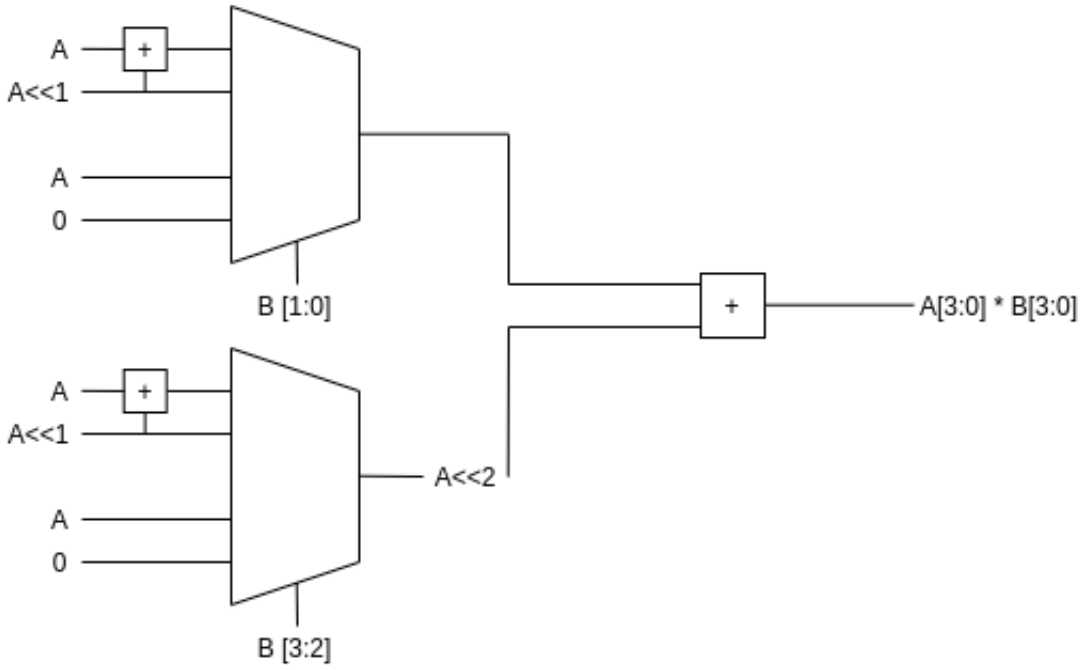

Figura 4: Multiplicador de 4 bits

Se puede observar que se realiza una manipulación previa a la entrada del mux, las cuales son equivalentes a multiplicaciones por números bajos (0-3). En el caso del mux inferior, la salida se desplaza dos veces, puesto que este es el resultado de los dos bits más significativos de B. Finalmente se suman las salidas.

4.3. Ejercicio 4

Con el mismo concepto del diagrama anterior, de utilizar circuitos de multiplicadores de 2 bits para construir de 4 bits se puede extender de 4 a 8, de 8 a 16 y finalmente de 16 a 32 bits. En las Figuras 5, 6 se diseñaron los multiplicadores base para alcanzar el multiplicador de 32 bits final de la Figura 7.

Para obtener el multiplicador final se requieren de 8 multiplicadores de 4 bits, que llevaría a 16 muxes.

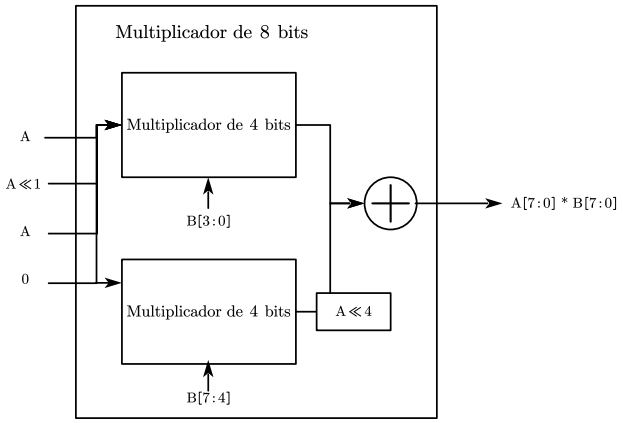

Figura 5: Multiplicador de 8 bits creado a partir de dos de 4 bits.

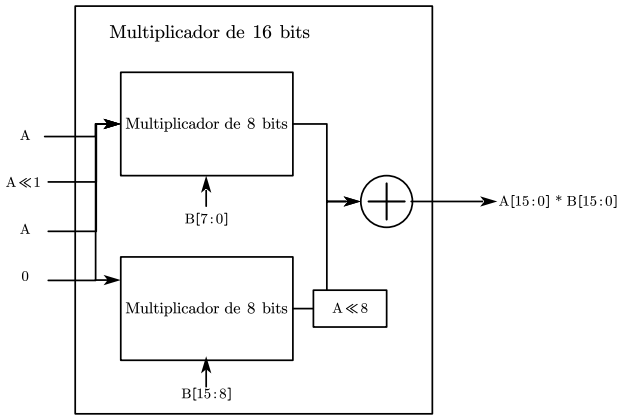

Figura 6: Multiplicador de 16 bits creado a partir de dos de 8 bits.

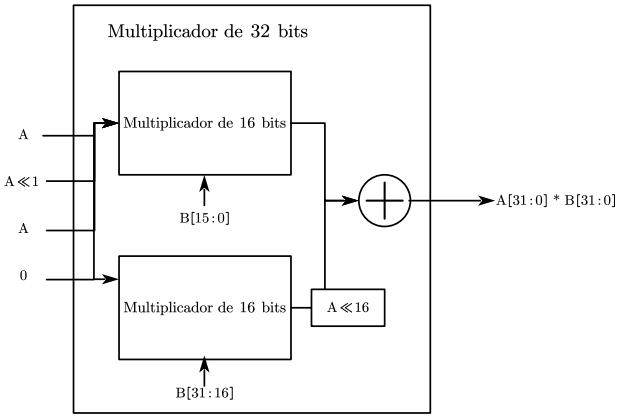

Figura 7: Multiplicador de 32 bits creado a partir de dos de 16 bits.

4.4. Ejercicio 5

Utilizando las recomendaciones del profesor, se escribió código 1 en Verilog para implementar el circuito presentado en la figura 8.

```
1  module sumador_wCarry( input bit1, input bit2, input carryIn, input
       result, input carryOut );
2      assign {carryOut, result} = bit1 + bit2 + carryIn;
3  endmodule
4
5  module array_multiplier_4b(input [3:0] A, input [3:0] B, input reset,
       output [7:0] result);
6
7      wire carry0, carry1, carry2, carry3, carry4, carry5, carry6,
         carry7, carry8, carry9, carry10, carry11, carry12, carry13;
```

```
8       wire sum_result2, sum_result3, sum_result6, sum_result4,
           sum_result7, sum_result8;
9
10      if(reset){
11          result = 'b00000000;
12      }
13      else{
14
15          sumador_wCarry suma0(
16              .bit1(A[0] & B[0]),
17              .bit2('b0),
18              .carryIn('b0),
19              .carryOut(),
20              .result(result[0])
21          );
22
23          sumador_wCarry suma1(
24              .bit1(A[1] & B[0]),
25              .bit2(A[0] & B[1]),
26              .carryIn('b0),
27              .carryOut(carry0),
28              .result(result[1])
29          );
30
31          sumador_wCarry suma2(
32              .bit1(A[1] & B[1]),
33              .bit2(A[2] & B[0]),
34              .carryIn(carry0),
35              .carryOut(carry1),
36              .result(sum_result2)
37          );
38
39          sumador_wCarry suma3(
40              .bit1(A[3] & B[0]),
41              .bit2(A[2] & B[1]),
42              .carryIn(carry1),
43              .carryOut(carry2),
44              .result(sum_result3)
45          );
46
```

```
47          sumador_wCarry suma4(
48              .bit1('b0),
49              .bit2(A[3] & B[1]),
50              .carryIn(carry2),
51              .carryOut(carry3),
52              .result(sum_result4)
53          );
54
55          sumador_wCarry suma5(
56              .bit1(sum_result2),
57              .bit2(A[0] & B[2]),
58              .carryIn('b0),
59              .carryOut(carry4),
60              .result(result[2])
61          );
62
63          sumador_wCarry suma6(
64              .bit1(sum_result3),
65              .bit2(A[1] & B[2]),
66              .carryIn(carry4),
67              .carryOut(carry5),
68              .result(sum_result6)
69          );
70
71          sumador_wCarry suma7(
72              .bit1(sum_result4),
73              .bit2(A[2] & B[2]),
74              .carryIn(carry5),
75              .carryOut(carry6),
76              .result(sum_result7)
77          );
78
79          sumador_wCarry suma8(
80              .bit1(carry3),
81              .bit2(A[3] & B[2]),
82              .carryIn(carry6),
83              .carryOut(carry7),
84              .result(sum_result8)
85          );
86          sumador_wCarry suma9(
```

```
87              .bit1(sum_result6),
88              .bit2(A[0] & B[3]),
89              .carryIn('b0),
90              .carryOut(carry8),
91              .result(result[3])
92          );
93
94          sumador_wCarry suma10(
95              .bit1(sum_result7),
96              .bit2(A[1] & B[3]),
97              .carryIn(carry8)),
98              .carryOut(carry9),
99              .result(result[4])
100         );
101
102         sumador_wCarry suma11(
103             .bit1(sum_result8),
104             .bit2(A[2] & B[3]),
105             .carryIn(carry9),
106             .carryOut(carry10),
107             .result(result[5])
108         );
109
110         sumador_wCarry suma12(
111             .bit1(carry7),
112             .bit2(A[3] & B[3]),
113             .carryIn(carry10),
114             .carryOut(result[7]),
115             .result(result[6])
116         );
117
118     }
119
120
121
122
123 endmodule
```

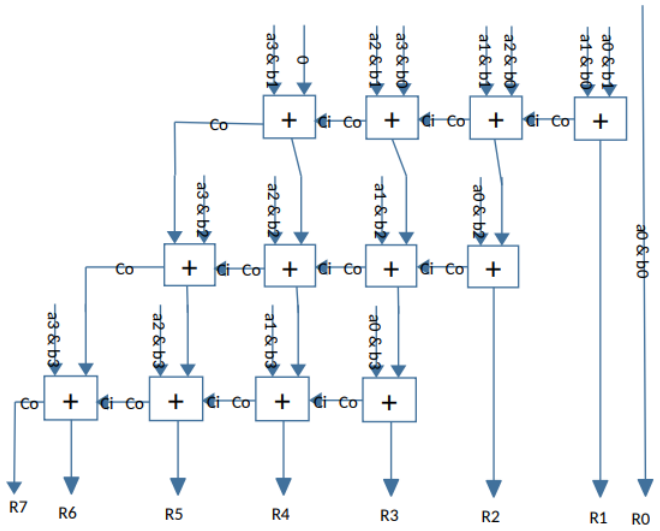

Figura 8: Array multiplier de 4 bits

4.5. Ejercicio 6

Analizando el array multiplier del ejercicio pasado, se puede visualizar este como un diagrama de bloques siguiendo sus entradas y salidas, como en la Figura 9.

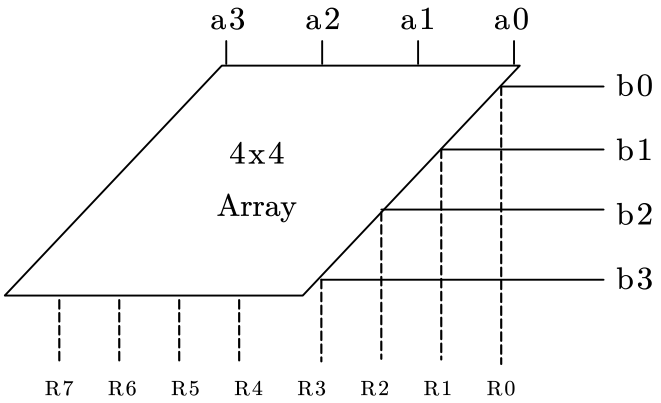

Figura 9: Diagrama de bloques de un array multiplier de 4 bits

Ahora, siguiendo el concepto de bloques funcionales, se pueden combinar 4 bloques de 4x4 para formar un bloque de 8x8, como en la Figura 10. Este bloque multiplica números de 8 bits para obtener un producto de 16 bits.

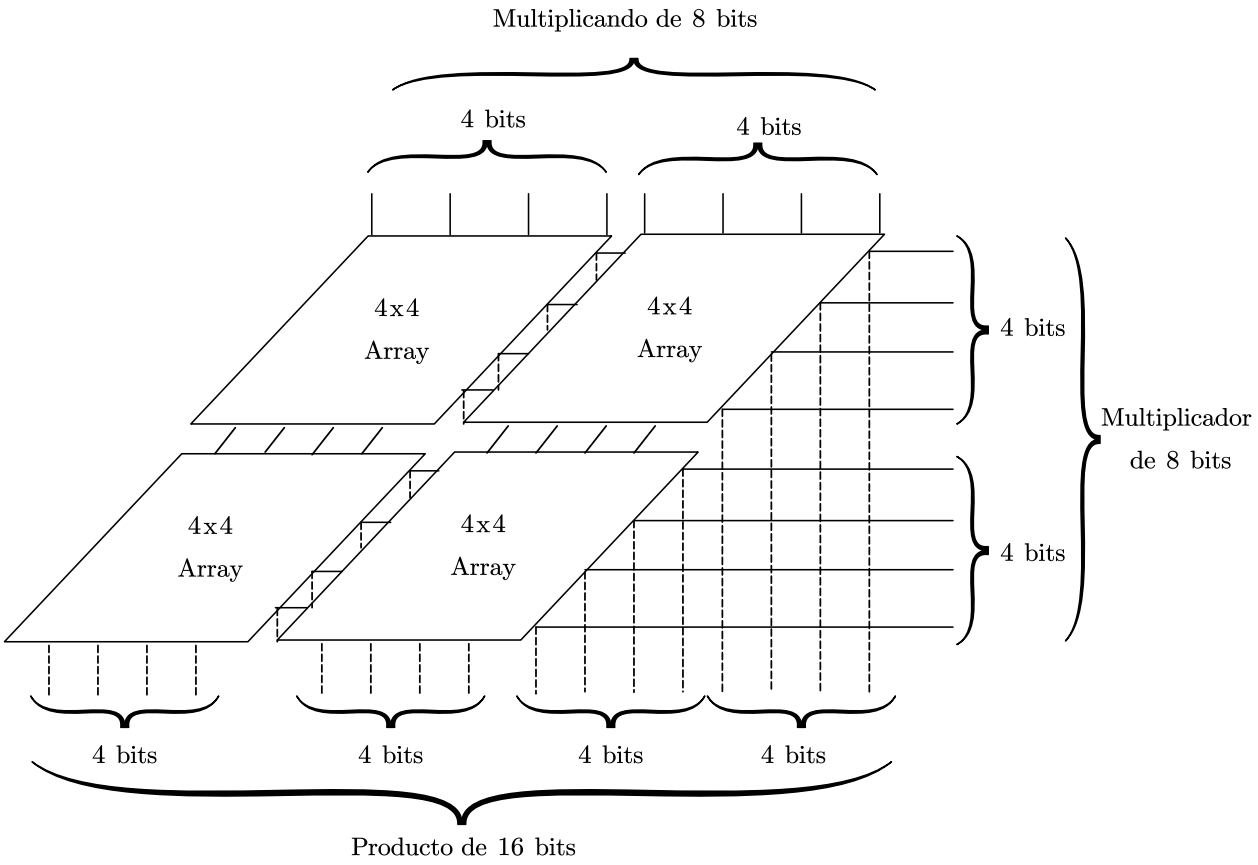

Figura 10: Diagrama de bloques de un array multiplier de 8 bits

Esta idea se puede extender para multiplicar números de n bits, entonces para obtener un array de nxn bits se sigue el diagrama de la Figura 11.

De esta forma, cambiando n = 32 se obtiene un array multiplier de 32 bits que obtendrá un producto de 64 bits, utilizaría 4 bloques de 16x16. Cada bloque de 16x16 ocupa 4 bloques de 8x8 y cada bloque de 8x8 ocupa 4 bloques de 4x4. Por lo que en resumen, el bloque de 32x32 requerirá 4^3 = 64 bloques de 4x4.

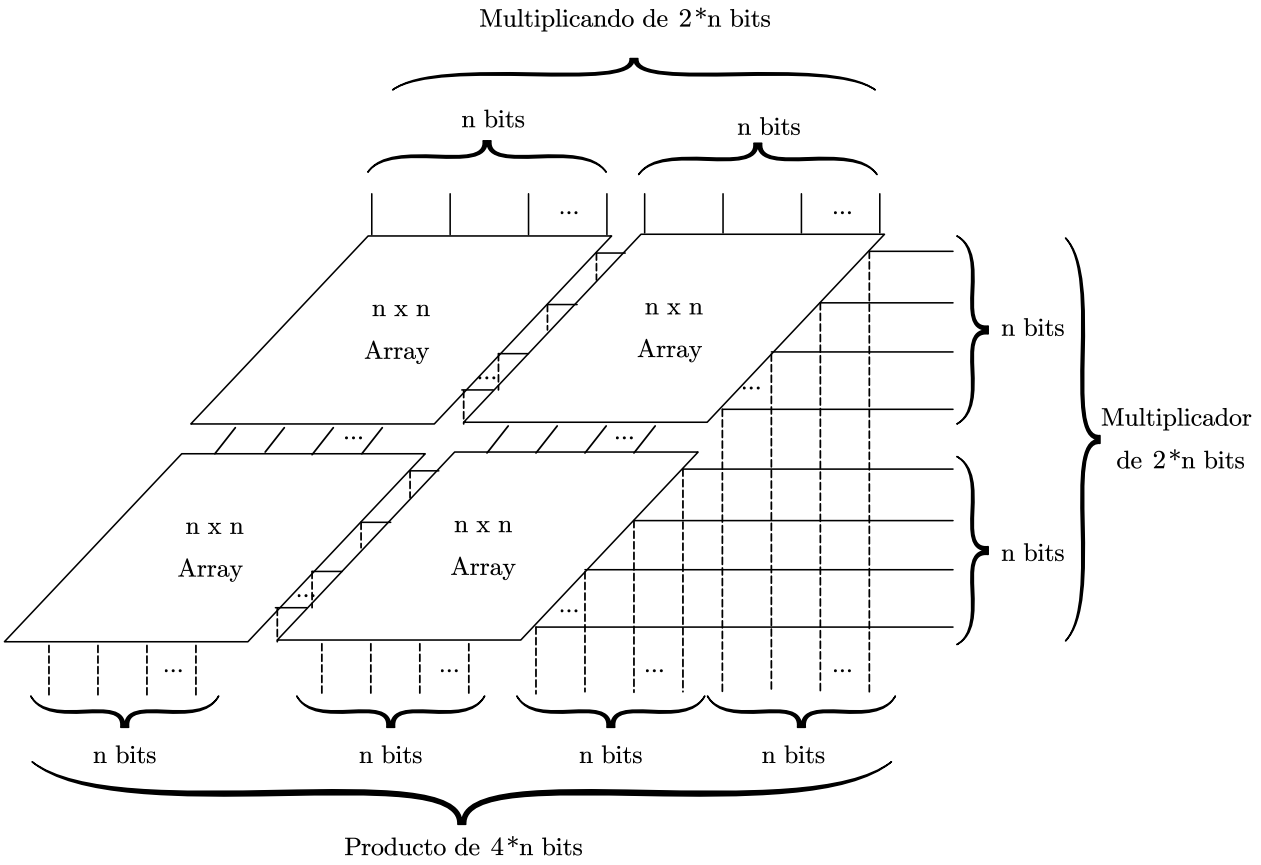

Figura 11: Diagrama de bloques de un array multiplier de n bits

4.6. Ejercicio 7

Para configurar el hardware para calcular el factorial se propone el siguiente código en Verilog:

```
module factorial(
        input wire [31:0] fuente,
        input wire control,
        output reg status,
        output [63:0] resultado,
        input reset,
        input clk
        );
reg [63:0] fact , aux, cnt;
reg [1:0] estado, prox_estado;
reg [31:0] A,B;

parameter ESPERANDO = 0;
```

```
14 parameter CALCULANDO = 1;
15 parameter FIN = 2;
16
17 generate
18         lut_multiplier_32b lut_32b(
19                 .A      (A),
20                 .B      (B),
21                 .M      (resultado)
22         );
23 endgenerate
24
25
26 always @ (posedge clk) begin
27     if (!reset) begin
28         estado <= ESPERANDO;
29         fact <= 1;
30         aux <= 1;
31     end else begin
32             estado <= prox_estado;
33             fact <= resultado;
34             aux <= cnt;
35     end
36 end
37
38 always @ (*) begin
39         A=fact;
40         cnt=1;
41         B=1;
42         status=0;
43         case (estado)
44                 ESPERANDO: begin
45                         A=1;
46                         if (control) prox_estado = CALCULANDO;
47                         else prox_estado = ESPERANDO;
48                 end
49                 CALCULANDO: begin
50                          B=aux;
51                          cnt=aux+1;
52                          if (aux==fuente) prox_estado = FIN;
53                          else prox_estado = CALCULANDO;
```

```
                end
                FIN: begin
                         status=1;
                        if (control) prox_estado = FIN;
                        else prox_estado = ESPERANDO;
                end
                default:
                        prox_estado = ESPERANDO;
        endcase
end
endmodule
```

Se observa que el código posee el construcción de **generate** para poder instanciar un módulo genérico del multiplier de 32 bits. Luego se pasa a la máquina de estados principal que calculará el factorial.

Seguido de esto, se crea el firmware que cumpla con las especificaciones propuestas se plantea hacer lo siguiente en pseudocódigo para cada número que se desea calcular el factorial:

```
#define FUENTE 0x0FFFFFF0
#define CONTROL 0x0FFFFFF4
#define RESULTADO 0x0FFFFFF8
#define STATUS 0x0FFFFFFC
#define FINAL 0x10000000
#define STOP 0
#define START 1

void main(){
    numero = 5;      //Numero a calcular factorial
    status = 0;      //Estatus

    // Escribe a CONTROL
    write(STOP, CONTROL);
    // Ingresa el numero a la fuente
    write(numero, FUENTE);

    write(START, CONTROL);

    // Loop de lectura para ver cuando termina
    do{
        status = read(STATUS);
    } while (status == 0);
```

```
24
25      // Ya termino, se escribe a FINAL
26      write(FINAL, read(RESULTADO));
27
28      status = 0;
29      write(STATUS, status);
30  }
```

Donde los métodos de `write` y `read` sólo escriben y leen de las posiciones de memoria definidas.

5. Observaciones y recomendaciones

- Se recomienda tener en cuenta el concepto de máquina de estados e implementar el código de Verilog de esta manera, por lo menos el de calcular el factorial. Puesto que esta máquina no obtendrá su resultado final sino hasta luego de varios ciclos de máquina.
- Se recomienda, casi que es indispensable, utilizar el bloque de **generate** de Verilog para crear bloques genéricos, por ejemplo en el Ejercicio 6 que se requieren 64 bloques de 4x4 array multipliers. Realizar estos bloques uno por uno manualmente, tardará un tiempo excesivo.
- Al traducir un circuito grande a Verilog, es conveniente utilizar nombres significativos de modo que los nodos en los que se realicen las conexiones sean adecuadas. Además, el uso de herramientas como autowire y autoinst, facilitan la creación de proyectos que poseen varias partes.

Referencias

[1] *7 Series FPGAs Data Sheet: Overview*. Xilinx®, 2020. URL: https://www.mouser.co.cr/datasheet/2/903/ds180_7Series_Overview-1591537.pdf.

[2] "NVIDIA Corporation". *GPU Gems 2: programming techniques for high-performance graphics and general-purpose*. 2.ª ed. Pearson Education, Inc., 2005. ISBN: 0128119055,9780128119051. URL: http://gen.lib.rus.ec/book/index.php?md5=58734DAE75228CD8CCF1BE995DA2893A.

[3] *FPGA Logic Cells Comparison*. 1-CORE Technologies. URL: http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf.

[4] P. Hollingworth. "Logic synthesis". En: *[Proceedings] EURO ASIC '90*. 1990, págs. 250-256. DOI: 10.1109/EASIC.1990.207949.

[5] *Nexys 4th FPGA Board Reference Manual*. Inf. téc. Digilent Inc., abr. de 2016.

[6] Douglas L Perry. *VHDL: Programming By Example, Fourth Edition*. 4.ª ed. McGraw-Hill, 2002.

[7] Zarrin Tasnim Sworna y col. "An Efficient Design of an FPGA-Based Multiplier Using LUT Merging Theorem". En: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, págs. 116-121. DOI: 10.1109/ISVLSI.2017.29.

[8] Frank Vahid. *Digital Design*. Wiley, 2011. URL: http://gen.lib.rus.ec/book/index.php?md5=78F6A70859399F8DA7A2A88FB2B9395B.

[9] Andrew Waterman y col. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Inf. téc. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, mayo de 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.