	Universidad de Costa Rica Escuela de ingeniería Eléctrica Laboratorio 3	EIE Escuela de Ingeniería Eléctrica
IE0424: Laboratorio de Circuitos Digitales II-2020		

Objetivo General

Utilizar la FPGA para el desarrollo de circuitos combinatorios.

Objetivos específicos

- Investigar el funcionamiento de la tarjeta de desarrollo FPGA Nexys 4.
- Utilizar las herramientas del Xilinx Vivado.
- Conocer y aplicar el flujo de diseño para sistemas basados en FPGA.

Anteproyecto

- Investigue acerca de los siguientes términos y realice una breve descripción del significado y uso de cada uno.
  - Síntesis de lógica
  - RTL
  - Place and route
  - Floorplan
  - FPGA
  - Slices de FPGAs
- Investigue cuál es la FPGA que tiene la tarjeta Nexys 4 DDR (nombre y número de parte) e investigue acerca de:
  - El número de celdas lógicas y slices que contiene.
  - Número máximo de pines de entrada y salida

Propuesta del problema.

Proceda ahora a obtener el código inicial del proyecto. Para ello, siga el siguiente enlace:

https://classroom.github.com/g/dbLEwLVG

De la misma forma que en el Laboratorio 1 y 2, autorice la aplicación, busque su nombre (si su nombre no aparece, contacte al profesor) y seleccione un equipo (si

su equipo no aparece listado, proceda a crear uno). Una vez finalizado un repositorio debió haber sido creado. Este repositorio va a ser el repositorio con el que va a trabajar y va a ser uno de los entregables de este laboratorio.

Proceda a clonar el repositorio que ha sido creado:

```
$ git clone https://github.com/ucr-ie-0424/...
```

La dirección completa de su repositorio debe aparecer seleccionando el botón verde que dice *Code.*

El código que se provee es el mismo código que se usó como punto de inicio del Laboratorio 1 y 2.

En este laboratorio se trabajará de nuevo con multiplicaciones y circuitos combinacionales. En particular, se trabajará con bloques generate en Verilog.

Ejercicio 1

De manera semejante al Laboratorio 2, escriba un firmware para se escriba en la dirección 0x10000000 el resultado del factorial de diferentes números: 5, 7, 10, 12. Llame a este firmware *firmware_lab3_part1.c*

Por defecto el compilador (GCC) no va a agregar instrucciones que son parte de la extensión RV32M.

Por tanto, modifique el *Makefile* de *firmware.c* para que se le pase el siguiente argumento a gcc:

```
-march=rv32im
```

Compile el nuevo firmware. Para ello, refresque el enlace simbólico de *firmware.c* y compile el código de nuevo:

```
$ src/firmware
$ ln -sf firmware_lab3_part1.c firmware.c
$ make
```

Verifique que su código ahora utiliza instrucciones de esta extensión. Para ello, puede utilizar la herramienta *objdump* de la misma manera que se hizo en el Laboratorio 1 y 2:

```
$ riscv32-unknown-elf-objdump -D firmware.elf
```

A diferencia del Laboratorio 2, proceda a realizar los siguientes pasos:

- Lance el proceso de síntesis e implementación para la tarjeta Nexys 4 y asegúrese de que estas terminan satisfactoriamente.
- Asimismo, utilice simulaciones postsíntesis para verificar su diseño.
- Anote la frecuencia que la herramienta de síntesis estima para esta parte, además del número de LUTs, Slices y Flip-Flops.

Ejercicio 2

En esta parte se va a implementar un multiplicador del tipo *array multiplier*.

Para ilustrar el funcionamiento de este multiplicador comience por repasar el algoritmo de multiplicación fundamental que aprendió en la escuela:

```
    1 2
x   1 3
    3 6
+ 1 2
  1 5 6
```

Esto mismo se puede escribir en binario como sigue:

```
      1 1 0 0
x     1 1 0 1
      1 1 0 0
    0 0 0 0
  1 1 0 0
+   1 1 0 0
  1 0 0 1 1 1 0 0
```

Como puede notar de la figura anterior, la mecánica de la multiplicación es la misma que en base 10. Note que a la hora de sumar se debe tomar en cuenta el acarreo.

Para utilizar el operador de suma de Verilog tomando el cuenta el acarreo puede hacer lo siguiente:

```
wire [n-1:0] wResult;
wire         wCarry;


assign {wCarry, wResult } = wA + wB;
```

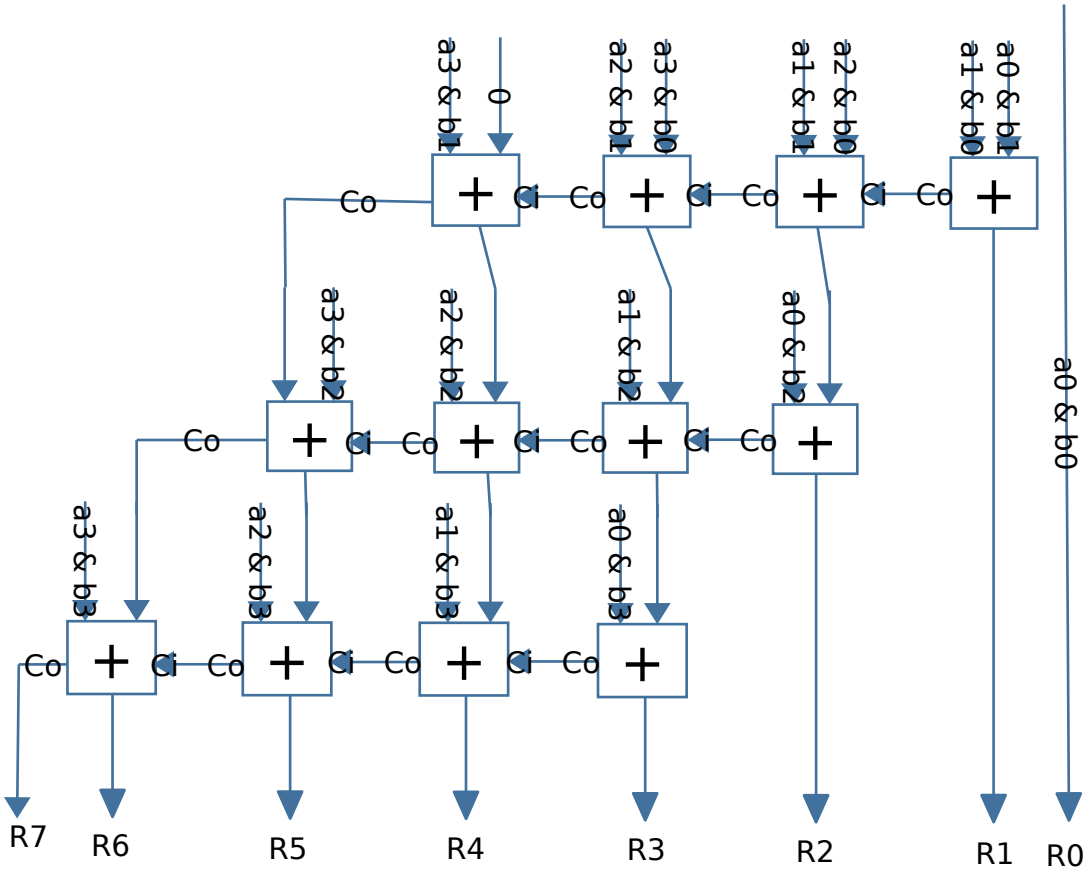
Sean dos números de 4 bits A = {a3,a2,a1,a0} y B = {b3,b2,b1,0} entonces note que la multiplicación se lleva a cabo de la siguiente manera:

				a3	a2	a1	a0
				x b3	b2	b1	b0
				a3b0	a2b0	a1b0	a0b0
			a3b1	a2b1	a1b1	a0b1	
		a3b2	a2b2	a1b2	a0b2		
	a3b3	a2b3	a1b3	a0b3			
R7	R6	R5	R4	R3	R2	R1	R0

De la figura anterior se puede observar lo siguiente:

- R0 = a0 & b0
- R1 = a1 & b0 + a0 & b1
- R2 = a2 & b0 + a1 & b1 + a0 & b2 + el acarreo de R1
- Etc…

A continuación se muestra una figura que ilustra la estructura del circuito que deberá implementar.



Siguiendo esta lógica implemente un módulo de Verilog que multiplique dos números de 4 bits sin signo. Para ello, instancie en system.v un módulo con el nombre arr_*multiplier_4b*. Este módulo debe de tener 3 entradas:

- Un número fuente a multiplicar de 4 bits.
- Un segundo número fuente a multiplicar de 4 bits.
- La señal de reset.

Adicionalmente debe de tener una salida de 8 bits:

- El resultado de la multiplicación de los dos números fuente de 4 bits.

Instancie el módulo dentro de su diseño de forma tal que se multiplique el contenido de las direcciones 0x0FFFFFF0 y 0x0FFFFFF4. El resultado se colocaría en la dirección 0x0FFFFFF8.

Vuelva a verificar la funcionalidad de su diseño usando un firmware que ahora escriba en la dirección 0x0FFFFFF0 cada uno de los números del 0 al 15. Por cada número que se escribió en la dirección anterior, se deberá escribir cada uno de los números del 0 al 15 en la dirección 0x0FFFFFF4. Esto con el fin de que se produzcan todas las multiplicaciones posibles con 2 números de 4 bits.

Para cada multiplicación el firmware debe de extraer el resultado de la multiplicación y escribir el resultado en la dirección 0x10000000 para que se vea reflejado en *out_byte*.

Llame a este firmware *firmware_lab3_part2.c*

Compile el nuevo firmware. Para ello, refresque el enlace simbólico de *firmware.c* y compile el código de nuevo:

```
$ src/firmware

$ ln -sf firmware_lab3_part2.c firmware.c

$ make
```

Proceda, asimismo a lanzar el proceso de síntesis e implementación para la tarjeta Nexys 4 y asegúrese de que su diseño no contenga warnings provocados por la lógica del módulo implementado anteriormente.

Asimismo, utilice simulaciones postsíntesis para verificar su diseño.

Anote la frecuencia que la herramienta de síntesis estima para esta parte, además del número de LUTs, Slices y Flip-Flops.

Ejercicio 3

Extienda el circuito del ejercicio anterior para multiplicar dos números de 32 bits.

Para implementar esto estudie la construcción de Verilog llamada ***generate***.

Los bloques de Verilog ***generate*** le permitirán ahorrar mucho tiempo y le enseñarán cómo escribir código genérico para una tarea que de otra forma puede resultar muy tediosa.

Para escribir los bloques *generate* puede usar el constructor `for` y pensar en una estructura con filas y columnas como la de la figura anterior.

Recuerde que puede declarar arreglos de cables. Observe el siguiente código (los puntos suspensivos son partes dejadas intencionalmente en blanco):

```
wire[2:0] wCarry[2:0];
genvar  CurrentRow,  CurrentCol;
generate
for ( CurrentCol = 0; CurrentCol < `MAX_COLS; CurrentCol = CurrentCol + 1)
begin : MUL_ROW
…
MODULE_ADDER  # (4) MyAdder
(
  .A( … ),
  .B( … ),
  .Ci(  wCarry[ CurrentRow ][ CurrentCol ] ),
  .Co( wCarry[ CurrentRow ][ CurrentCol + 1]),
);
…
end
endgenerate
```

Además recuerde que `Ci` de los sumadores de las columnas de la izquierda son cero.

```
assign wCarry [CurrentRow ][ 0 ] = 0;
```

Siguiendo esta lógica implemente un módulo de Verilog que multiplique dos números de 32 bits sin signo. Para ello, instancie en system.v un módulo con el nombre arr_*multiplier_32b*. Este módulo debe de tener 3 entradas:

- Un número fuente a multiplicar de 32 bits.
- Un segundo número fuente a multiplicar de 32 bits.
- La señal de reset.

Adicionalmente debe de tener una salida de 64 bits:

- El resultado de la multiplicación de los dos números fuente de 32 bits.

Instancie el módulo dentro de su diseño de forma tal que se multiplique el contenido de las direcciones 0x0FFFFFF0 y 0x0FFFFFF4. El resultado se colocaría en la direcciones 0x0FFFFFF8 (32 bits menos significativos) y 0x0FFFFFFC (32 bits más significativos).

Vuelva a verificar la funcionalidad de su diseño usando un firmware que ahora escriba en las direcciones 0x0FFFFFF0 y 0x0FFFFFF4 lo siguinte:

- Escribir 25 en 0x0FFFFFF0.
- Escribir 7 en 0x0FFFFFF4.
- Escribir 635 en 0x0FFFFFF0.
- Escribir 1023 en 0x0FFFFFF4.
- Escribir 2157297371 en 0x0FFFFFF0.
- Escribir 562 en 0x0FFFFFF4.
- Escribir 9813723 en 0x0FFFFFF0.
- Escribir 4036341403 en 0x0FFFFFF4.
- Escribir 3628800 en 0x0FFFFFF4.
- Escribir 1 en 0x0FFFFFF0.
- Escribir 4068839099 en 0x0FFFFFF0.
- Escribir 0 en 0x0FFFFFF0.

Posteriormente firmware debe de extraer el resultado de la multiplicación, contar cuántos bits están en 1 y poner el resultado en la dirección 0x10000000 para que se vea reflejado en *out_byte.*

Llame a este firmware *firmware_lab3_part3.c*

Compile el nuevo firmware. Para ello, refresque el enlace simbólico de *firmware.c* y compile el código de nuevo:

```
$ src/firmware
$ ln -sf firmware_lab3_part3.c firmware.c
$ make
```

Proceda, asimismo a lanzar el proceso de síntesis e implementación para la tarjeta Nexys 4 y asegúrese de que su diseño no contenga warnings provocados por la lógica del módulo implementado anteriormente.

Asimismo, utilice simulaciones postsíntesis para verificar su diseño.

Anote la frecuencia que la herramienta de síntesis estima para esta parte, además del número de LUTs, Slices y Flip-Flops. ¿Cómo se compara con el Ejercicio 1 y 2?

Por último, responda las siguientes preguntas:

- ¿Son los bloques **generate** sintetizables?

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