	Universidad de Costa Rica Escuela de ingeniería Eléctrica Laboratorio 2	EIE Escuela de Ingeniería Eléctrica
IE0424: Laboratorio de Circuitos Digitales I-2021		

Objetivo General

Usar Vivado para el desarrollo de circuitos combinatorios.

Objetivos específicos

- 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.

Anteproyecto.

- Investigue qué instrucciones forman parte de la extensión RV32M y describa cada una de estas instrucciones brevemente.
- Describa ¿cómo se definen y cómo se usan los parámetros de instancias en módulos implementados en el lenguaje Verilog.
- Investigue qué es una LUT (look up table).
- Investigue qué es el alineamiento de memoria y para qué sirve.
- Investigue acerca de los siguientes términos y realice una breve descripción del significado y uso de cada uno.
    - Síntesis de lógica
    - RTL (register transfer level)
    - 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/t-gjjtJC
```

De la misma forma que en el Laboratorio 1, 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.

En este laboratorio se trabajará con multiplicaciones. En particular, se calcularán factoriales. Recuerde, que el factorial de un número se puede definir como:

$$n! = 1 \cdot 2 \cdot 3 ... (n-2) \cdot (n-1) \cdot n$$

La implementación de RISC-V que se usó en el Laboratorio 1 es capaz de hacer multiplicaciones si se le habilita el parámetro *ENABLE_MUL*. Para ello, proceda a agregar este parámetro en la instancia de picorv32.

Este parámetro habilita instrucciones de multiplicación que son parte de las extensión RV32M.

Ejercicio 1 (Obligatorio) (5%)

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_lab2_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_lab2_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:

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

Ahora realice cambios a system.v para que tenga una salida adicional de 32 bits que se llame *out_fact*. Esta salida va a presentar los datos que el firmware intente escribir a la dirección 0x10000000 pero a diferencia de *out_byte*, esta debe de presentar los 32 bits completos que se le escriben a esa dirección (0x10000000).

Muestre los resultados mediante alguna simulación que demuestre el correcto cálculo del factorial de los números escogidos.

Asimismo, calcule de la simulación cuántos ciclos de reloj en promedio le toma al CPU calcular cada uno de los diferentes factoriales.

Ejercicio 2 (Obligatorio) (5%)

Usando el código del Ejercicio 1 (mismo firmware y mismo diseño), proceda a lanzar 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 3 (Obligatorio) (5%)

Se va implementar un multiplicador con un algoritmo distinto. Para ello, instancie en system.v un módulo con el nombre *lut_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.

Para entender este algoritmo debe recordar una tabla de multiplicar como la siguiente:

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	9

Esta tabla puede estar almacenada en una memoria, como por ejemplo una ROM, y es lo que se conoce como una LUT (look up table).

Buscar el resultado de una multiplicación en una LUT es muy rápido, sin embargo, se vuelve poco práctico cuando hay que multiplicar números muy grandes.

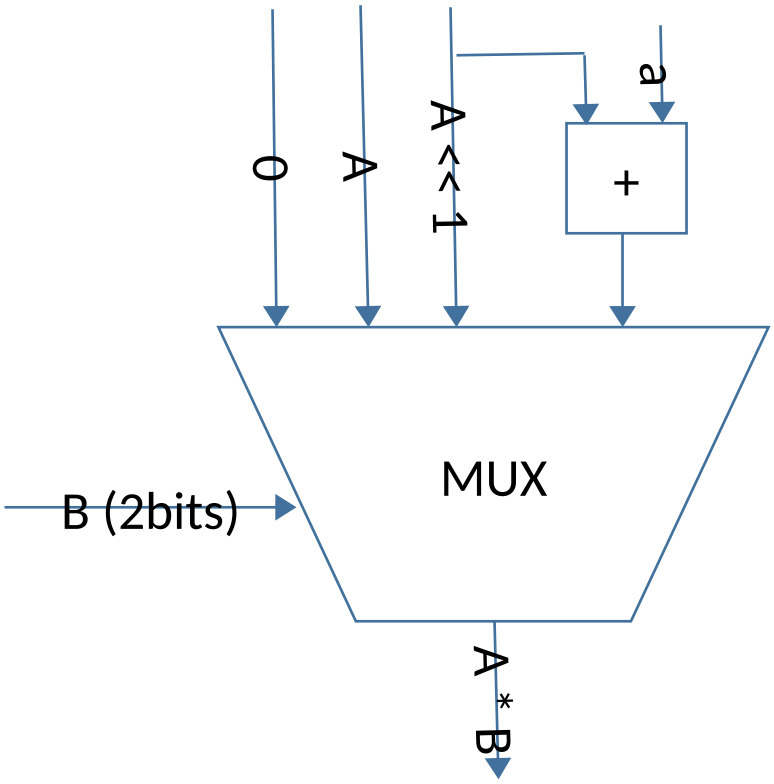
¿Cuantas filas y columnas tendría una LUT que permita multiplicar dos números de 32 bits?

Por otro lado se pueden combinar pequeñas tablas LUT como la anterior para obtener el resultado de números con muchos bits.

Para dos números de 2 bits A y B la tabla anterior se puede representar de la siguiente manera:

B	A*B	Descripción
0 (00)	0	Sin importar el valor de A, si B es cero A*B es cero.
1 (01)	A	Sin importar el valor de A, si B es uno A*B es A.
2 (10)	2*A	Multiplicar por 2 es un corrimiento a la izquierda, A << 1
3 (11)	3*A	Multiplicar por 3 es un corrimiento a las izquierda mas A, (A << 1) + a

Esto ultimo se acostumbra implementar como un multiplexor como se muestra a continuación:



En este momento se deberá preguntar, este MUX permite multiplicar dos 2 números de 2 bits, pero ¿qué ocurre con números de más de 2 bits?

Note que la figura anterior aplica para B de 2 bits, pero no importa el número de bits que tenga A.

Colocando varios bloques de multiplexores como el que se muestra y sumando los resultados parciales corridos a la izquierda el número apropiado de posiciones, se pueden multiplicar números de cualquier tamaño.

Haga un diseño de cómo conectar varios multiplexores que permita primeramente multiplicar dos números de 4 bits. Como una pista recuerde que:

$A * B = A * (b3 * 2^3 + b2 * 2^2 + b1 * 2^1 + b0 * 2^0)$

$= (A*b3 * 2^3 + A*b2 * 2^2 + A*b1 * 2^1 + A*b0 * 2^0)$

Recuerde que 2^n es un corrimiento a la izquierda n posiciones. Piense cómo agrupar lo anterior en grupos de 2 bits y acomodarlo en sumadores en cascada.

Implemente un circuito de multiplicación para multiplicar dos números de 4 bits para comenzar.

Escriba un firmware para escriba los siguientes números en las siguientes direcciones:

- Escribir 1 en 0x10000004.
- Escribir 2 en 0x10000008.
- Escribir 2 en 0x10000004.
- Escribir 3 en 0x10000008.

- Escribir 6 en 0x10000004.
- Escribir 4 en 0x10000008.

Llame a este firmware *firmware_lab2_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_lab2_part3.c firmware.c

$ make
```

Modifique *system.v* para que lea los datos que firmware escribe en 0x10000004 y 0x10000008 le pase esos números como fuentes a su instancia *lut_multiplier_4b.*

Verifique que los resultado que su módulo obtiene son correctos mediante una simulación.

Ejercicio 4 (Obligatorio) (30%)

Utilizando la técnica de multiplicación del Ejercicio 3, implemente un circuito de multiplicación para que tome como fuente dos número de 32 bits. Para ello, instancie en system.v un módulo con el nombre *lut_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.

Vuelva a verificar la funcionalidad de su diseño usando un firmware que ahora escriba en la dirección 0x1000000C y 0x10000010, los números:

- Escribir 120 en 0x1000000C.
- Escribir 2 en 0x10000010.
- Escribir 19 en 0x1000000C.
- Escribir 17 en 0x10000010.
- Escribir 3628800 en 0x1000000C.
- Escribir 11 en 0x10000010.
- Escribir 39916800 en 0x1000000C.
- Escribir 12 en 0x10000010.
- Escribir 3628800 en 0x10000010.

Llame a este firmware *firmware_lab2_part4.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_lab2_part4.c firmware.c
$ make
```

Modifique *system.v* para que lea los datos que firmware escribe en 0x1000000C y 0x10000010 y le pase esos números como fuentes a su instancia *lut_multiplier_32b.*

Verifique que los resultado que su módulo obtiene son correctos mediante una simulación.

Ejercicio 5 (Obligatorio) (5%)

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
```

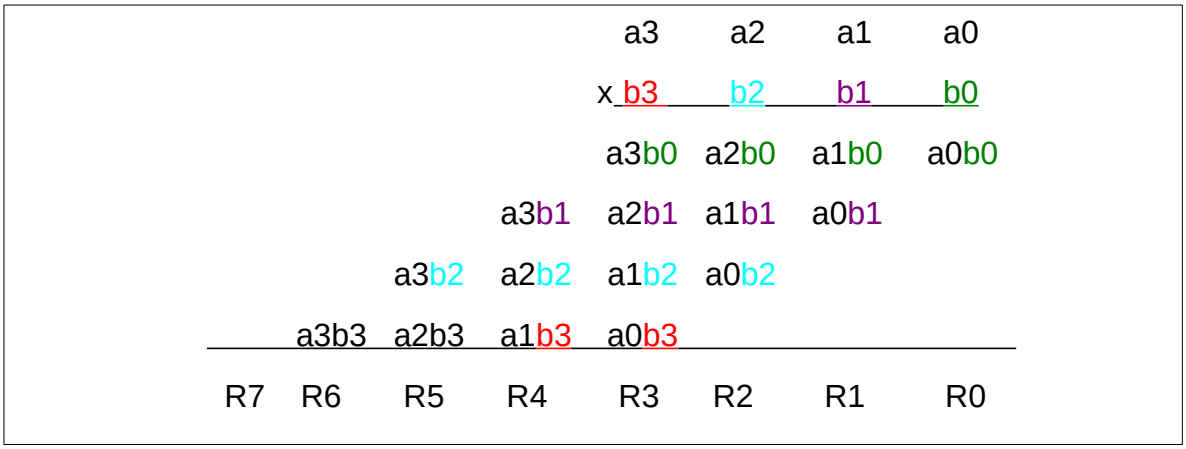
Como puede notar de la figura anterior, la mecánica de la multiplicación es la misma que en base 10. Note que a la hora de sumar se debe tomar en cuenta el acarreo.

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;
```

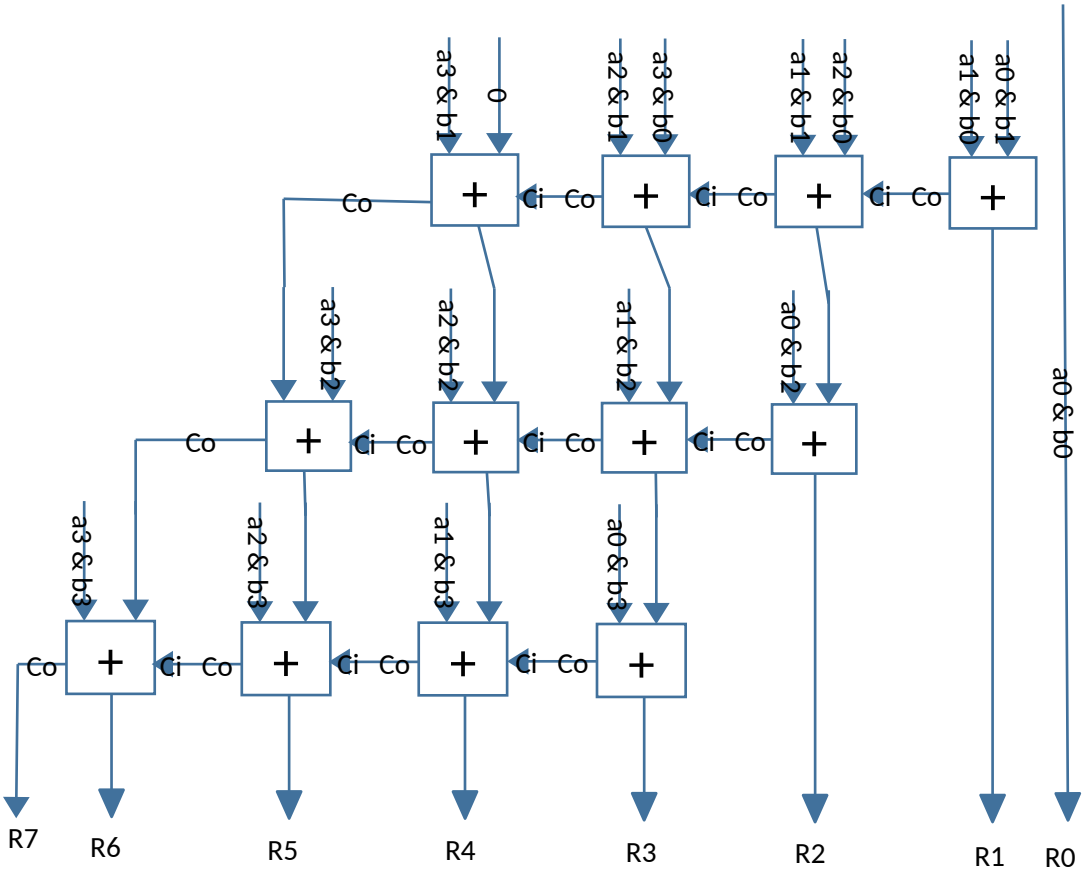
Sean dos números de 4 bits A = {a3,a2,a1,a0} y B = {b3,b2,b1,0} entonces note que la multiplicación se lleva a cabo de la siguiente manera:



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_lab2_part5.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_lab2_part5.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 6 (Obligatorio) (30%)

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_lab2_part6.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_lab2_part6.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?

Ejercicio 7 (Obligatorio) (20%)

Utilice la implementación de LUTs o array mulriplier para multiplicar dos números de 32 bits para calcular factoriales. Para ello, utilice la dirección 0x0FFFFFF0 para colocar el número al cual se le quiere calcular el factorial. El resultado del factorial se colocaría en la dirección 0x0FFFFFF8.

Para esta parte el cálculo del factorial puede tomar varios ciclos de reloj. Para ello, haga que su diseño use un registro de control para empezar a calcular el factorial. Para ello, puede usar la dirección 0x0FFFFFF4, de tal forma que si se le escribe un 1, su diseño deberá empezar a calcular el factorial.

Asimismo, cuando se tenga calculado el resultado, su diseño escribirá un 1 a la dirección 0x0FFFFFFC (registro de status). Esto le permitirá al programador saber cuándo el resultado que se colocará en la dirección 0x0FFFFFF8 es válido.

Escriba un firmware que use las direcciones anteriores para calcular el factorial de los mismos números del Ejericicio 1. Note que esta vez su código, después de mandar a calcular el factorial, deberá de monitorear el registro de status (dirección

0x0FFFFFFC) para saber cuándo el resultado está listo y luego leer el resultado correcto para desplegarlo en la dirección 0x10000000.

Por ejemplo, el firmware para calcular el factorial de 5 deberá de:

1. Escribir 0 en 0x0FFFFFF4 para indicar en el registro de control que aún no empiece a calcular ningún factorial.
2. Escribir 5 en 0x0FFFFFF0. Este sería el factorial a calcular.
3. Escribir 1 en 0x0FFFFFF4 para indicar en el registro de control se empiece a calcular el factorial.
4. Leer la dirección 0x0FFFFFFC. Si el valor obtenido es 0 ejecutar el paso 4 de nuevo. Si el valor obtenido es 1 seguir ejecutando el paso 5.
5. Leer la dirección 0x0FFFFFF8. Este registro tendrá el resultado final.
6. Poner el valor obtenido en el paso 5 en la dirección 0x10000000.

Llame a este firmware *firmware_lab1_part7.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_lab2_part7.c firmware.c

$ make
```

Realice simulaciones para los números que escogió y calcule de la simulación cuántos ciclos de reloj en promedio le toma al CPU calcular cada uno de los factoriales.

Compare los resultados con los obtenidos en el Ejercicio 1.