Universidad de Costa Rica

Bitácora

Experimento 2

Prof. José Daniel Hernández

Luis Soto Camacho, B57082
Yeison Rodríguez Pacheco, B56074

Grupo 2

20 de septiembre de 2019

Índice

Índice de figuras

1. Objetivos

1.1. Objetivo general

- Utilizar la FPGA para el desarrollo de circuitos combinatorios

1.2. 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

2. Correcciones realizadas al anteproyecto y a la solución propuesta en el mismo

No se realizaron correcciones al anteproyecto, ya que se contempló todo lo necesario para la realización de este experimento.

3. Resultados, justificación y preguntas de la guía del laboratorio

3.1. Ejercicio 1

Comenzando, se añadió el argumento -march=rv32im al Makefile del firmware, de forma que se habilite el uso de las instrucciones de la extensión RV32M, la cual posee instrucciones de multiplicación y división. El código del makefile queda como se ve a continuación:

```
all: firmware.S firmware.c firmware.lds
$(TOOLCHAIN_PREFIX)gcc -Os -ffreestanding -march=rv32im -nostdlib -o
firmware.elf firmware.S firmware.c \
 --std=gnu99 -Wl,-Bstatic,-T,firmware.lds,-Map,firmware.map,--strip-debug -lgcc
$(TOOLCHAIN_PREFIX)objcopy -O binary firmware.elf firmware.bin
python3 ./makehex.py firmware.bin 4096 > firmware.hex
```

Como se puede ver, la opción ya está añadida, posteriormente, se agrega una función que calcule el factorial del valor ingresado en el código del firmware, y en este mismo se calcula y se guarda el valor obtenido, el código del firmware queda como se ve a continuación:

```
static void putuint(uint32_t i) {
*((volatile uint32_t *)0x10000000) = i;
}

uint32_t factorial(int n) {
if(n == 0) {
return 1;
}
if (n == 1){
return 1;
}
else{
return n * factorial(n-1);
}
}
```

```
void main() {
uint32_t fact_n = factorial(4);
putuint(fact_n);
while (1) { // para iterar infinitamente
}
}
```

En el código de verilog del sistema, en la instanciación del picorv32 se envía un parametro ENABLE_MUL = 1, de forma que se permita el uso de las mismas en el procesador, con esto se compila el firmware, y se usa la siguiente instrucción para desemsablar el binario obtenido:

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

Con esto se consigue la siguiente línea de texto, ya sea para el cálculo de cualquier número que se le ingrese:

```
Disassembly of section .memory:

00000000 <factorial-0x10>:
   0:   00004137                lui     sp,0x4
   4:   00010113                mv      sp,sp
   8:   02c000ef                jal     ra,34 <main>
   c:   00100073                ebreak

00000010 <factorial>:
  10:   00050793                mv      a5,a0
  14:   00100693                li      a3,1
  18:   00100513                li      a0,1
  1c:   00078713                mv      a4,a5
  20:   00f6f863                bgeu    a3,a5,30 <factorial+0x20>
  24:   fff78793                addi    a5,a5,-1
  28:   02e50533                mul     a0,a0,a4
  2c:   ff1ff06f                j       1c <factorial+0xc>
  30:   00008067                ret

00000034 <main>:
  34:   ff010113                addi    sp,sp,-16 # 3ff0 <end+0x3f8c>
  38:   00300513                li      a0,3
  3c:   00112623                sw      ra,12(sp)
  40:   fd1ff0ef                jal     ra,10 <factorial>
  44:   100007b7                lui     a5,0x10000
  48:   00a7a023                sw      a0,0(a5) # 10000000 <end+0xfffff9c>
  4c:   0000006f                j       4c <main+0x18>
  50:   3a434347                fmsub.d ft6,ft6,ft4,ft7,rmm
  54:   2820                    fld     fs0,80(s0)
  56:   29554e47                fmsub.s ft8,fa0,fs5,ft5,rmm
  5a:   3820                    fld     fs0,112(s0)
  5c:   322e                    fld     ft4,232(sp)
  5e:   302e                    fld     ft0,232(sp)
  60:   0000                    unimp
```

Figura 1: Código del factorial desensamblado. Ejercicio 1

Como se puede ver, en en código obtenido, en la función de factorial se tiene la instrucción mul, la cual fue habilitada anteriormente con la extensión RV32M, por lo que se considera que esto se logró exitosamente. Una vez hecho esto, se usa la función factorial para calcular el de tres números, para esto se comienza calculando el factorial de 4:

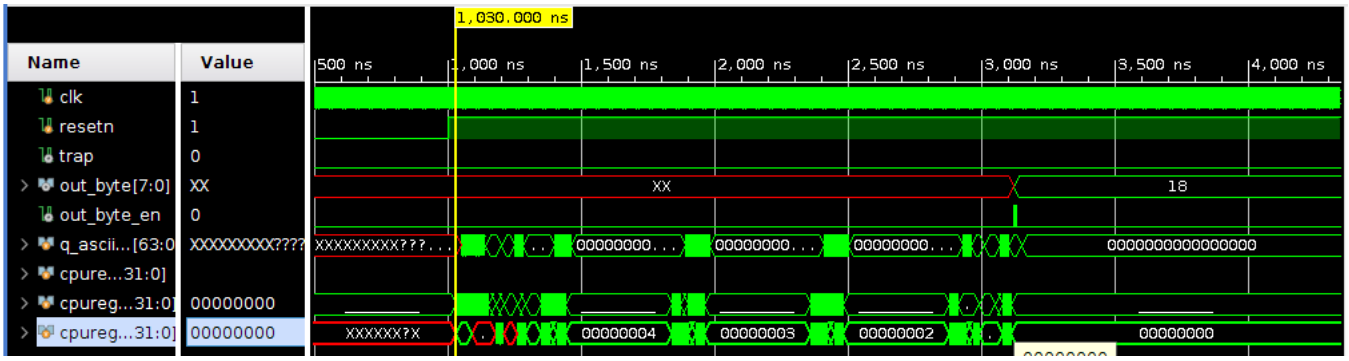

Figura 2: Inicio del cálculo del factorial de 4. Ejercicio 1

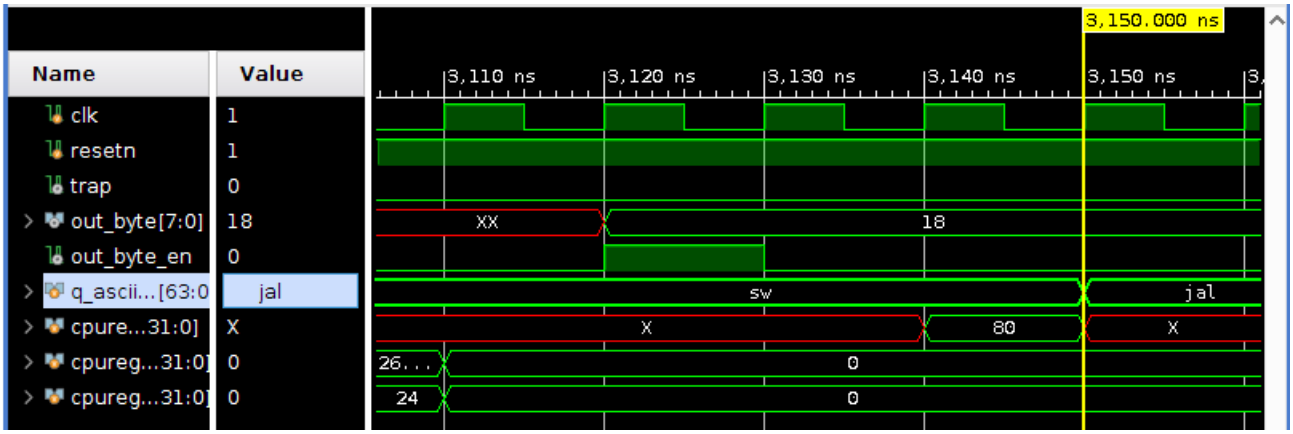

Figura 3: Final del cálculo del factorial de 4 y su resultado. Ejercicio 1

De las figuras anteriores se puede ver que el tiempo inicial es de 1.03 ns y el final de 3.15 ns, por lo que el tiempo total de cálculo es de 2.12 ns, el ciclo de reloj del procesador es de 0.01 ns, por lo que la cantidad de ciclos que se tarda el cálculo es de 212 ciclos, en la segunda figura, se puede observar el valor 24 en decimal del registro cpuregs_wrdata, el cual es el resultado del factorial de 4, por lo cual este cálculo se considera exitoso. Se continúa con el cálculo del factorial de 6:

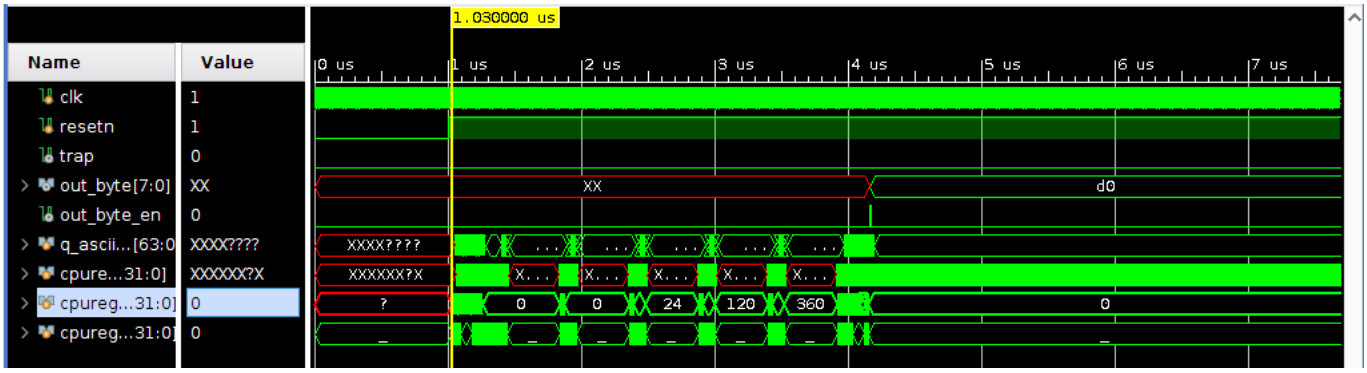

Figura 4: Inicio del cálculo del factorial de 6. Ejercicio 1

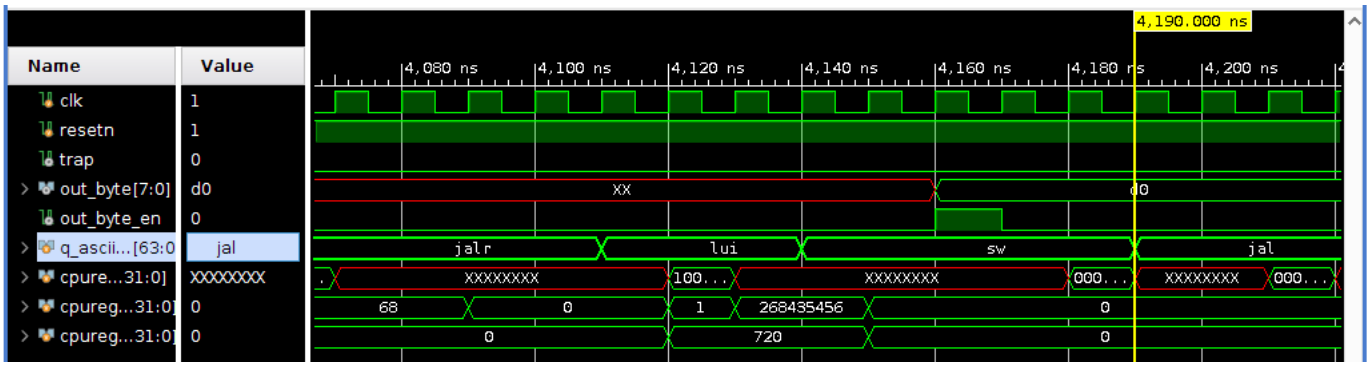

Figura 5: Final del cálculo del factorial de 6 y su resultado. Ejercicio 1

El inicio del cálculo se tiene en 1.03 ns, mientras que el final se da en 4.19 ns, por lo que el tiempo de cálculo es de 3.16 ns, por lo que se tiene que este se tarda 316 ciclos de reloj, y el resultado se tiene de 720 en decimal en el mismo registro, por último se procede a calcular el factorial de 12:

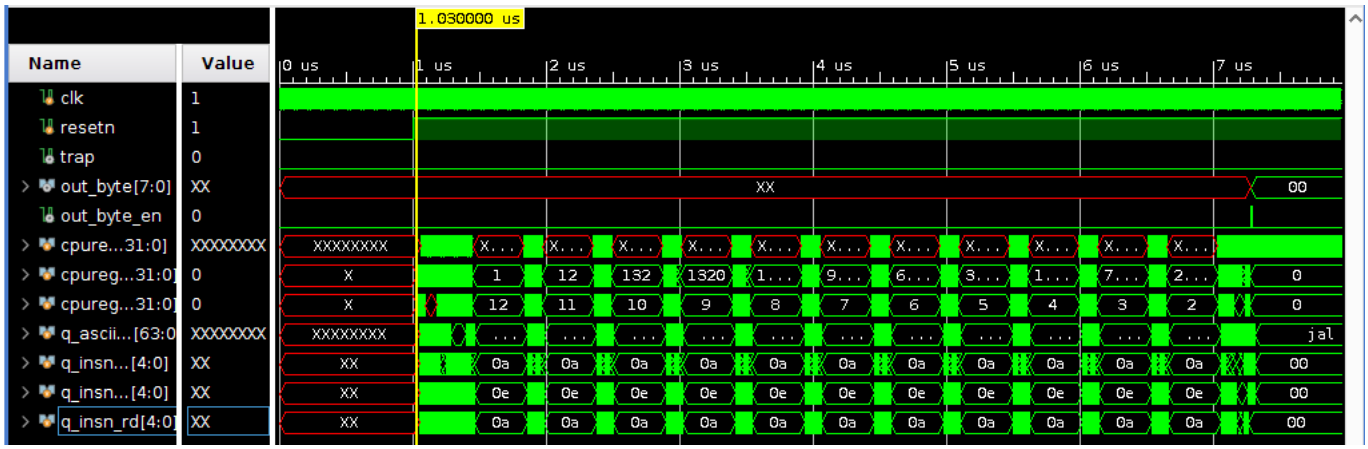

Figura 6: Inicio del cálculo del factorial de 12. Ejercicio 1

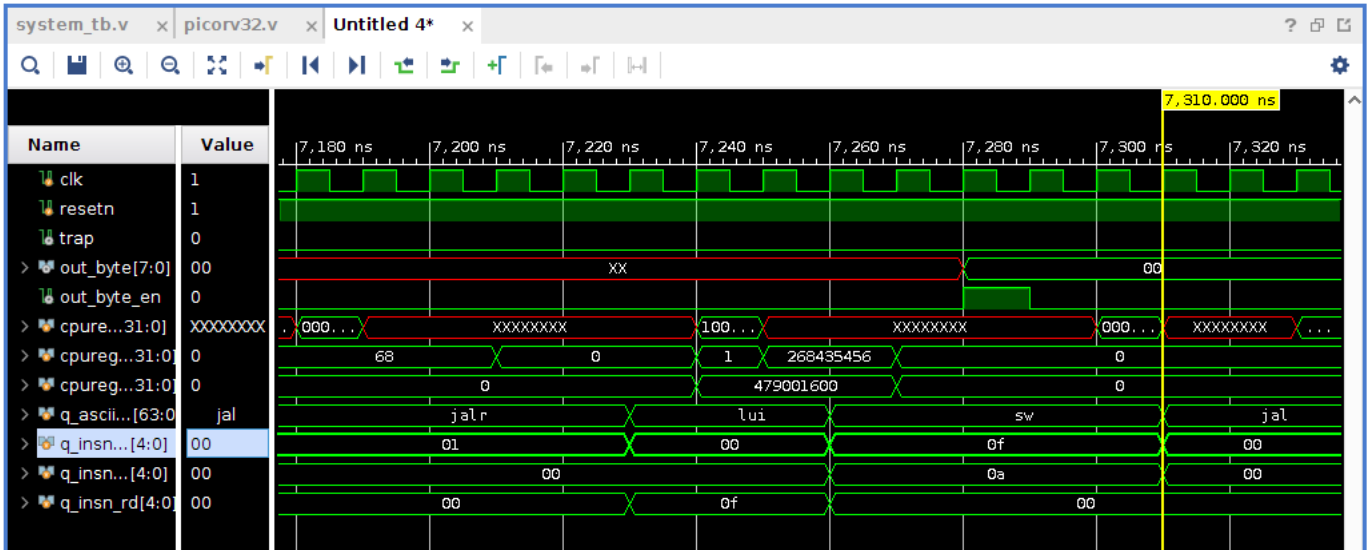

Figura 7: Final del cálculo del factorial de 12 y su resultado. Ejercicio 1

El inicio del cálculo está en 1.03 ns y el final está en 7.31 ns, por lo que el tiempo de cálculo es de 6.28 ns y los ciclos de reloj que tarda son 628, el resultado del factorial es de 479001600 en decimal y se tiene en el mismo registro que las simulaciones anteriores.

Por último, se tiene el reporte de síntesis obtenido:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1193	0	63400	1.88
LUT as Logic	1145	0	63400	1.81
LUT as Memory	48	0	19000	0.25
LUT as Distributed RAM	48	0		
LUT as Shift Register	0	0		
Slice Registers	795	0	126800	0.63
Register as Flip Flop	795	0	126800	0.63
Register as Latch	0	0	126800	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

Figura 8: Reporte de resultado de la síntesis. Ejercicio 1

3.2. Ejercicio 2

En el ejercicio 2 se realizó un multipicador de dos números de 4 bits, para esto se requirió un módulo sumador de dos bits y un carry que presente como salida la suma y el carry. A continuación se presenta el módulo sumador:

```
module sumador (
    input wA,
    input wB,
    input in_carry,
    output wResult,
    output wCarry
);
    assign {wCarry, wResult} = wA + wB + in_carry;
endmodule
```

Realizando múltiples instancias (en este caso 12) de los módulos sumadores de 1 bit se puede realizar una multiplicación, ya que el algoritmo de la misma lo permite. A continuación se presentan las intancias de módulos sumadores que calculan la multiplicación:

```
wire [11:0] c;
wire [5:0] ld; //line down
assign out_byte[0] = a[0] & b[0];
sumador R1(a[0] & b[1], a[1] & b[0],1'b0, out_byte[1],c[0]);
sumador x0(a[2] & b[0] ,a[1] & b[1] , c[0] ,ld[0] , c[1] );
sumador x1(a[3] & b[0], a[2] & b[1] ,c[1] ,ld[1], c[2]);
sumador x2( 1'b0 , a[3] & b[1] , c[2] , ld[2] , c[3]);
sumador R2( ld[0] , a[0] & b[2] , 1'b0 , out_byte[2] , c[4] );
sumador y0( ld[1] , a[1] & b[2] , c[4] , ld[3] , c[5] );
sumador y1( ld[2] , a[2] & b[2] , c[5] , ld[4] , c[6] );
sumador y2( a[3] & b[2] , c[3] , c[6] , ld[5] , c[7] );
sumador R3( ld[3] , a[0] & b[3] , 1'b0 , out_byte[3] , c[8] );
sumador R4( ld[4], a[1] & b[3] , c[8] , out_byte[4] , c[9] );
sumador R5( ld[5] , a[2] & b[3] , c[9] , out_byte[5] , c[10] );
sumador R6( c[7] , a[3] & b[3] , c[10] , out_byte[6] , out_byte[7] );
```

Este circuito se implementó en system.v, dentro del firmware se ingresaron dos números, que eran 4 y 5, esta implementación se encarga de interceptar estos valores, multiplicarlos y guardarlos en una dirección de memoria. En la figura 9 se presenta la simulación. Como podemos ver en la simulación, el resultado es 14 en hex (20 en decimal) que vemos que es correcto. También vemos que este número se guarda en la dirección 0x0FFFFFF8.

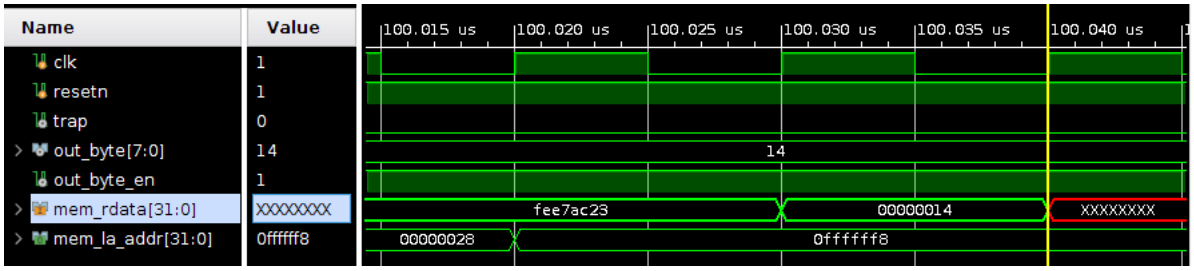

Figura 9: Multiplicación de dos números de 4 bits (4 y 5). Ejercicio 2

En la figura 10 vemos un reporte de la cantidad de slice de lógica combinacional y secuencial.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	3165	0	63400	4.99
LUT as Logic	1069	0	63400	1.69
LUT as Memory	2096	0	19000	11.03
LUT as Distributed RAM	2096	0		
LUT as Shift Register	0	0		
Slice Registers	547	0	126800	0.43
Register as Flip Flop	547	0	126800	0.43
Register as Latch	0	0	126800	0.00
F7 Muxes	1088	0	31700	3.43
F8 Muxes	539	0	15850	3.40

Figura 10: Reporte de slices, ejercicio 2.

3.3. Ejercicio 3

En el ejercicio 3 se realizó el multiplicador del ejercicio anterior instanciando los blosque sumadores de 1 bit pero utilizando el bloque generate. A continuación se presenta el código del generate para realizar este cálculo.

```
parameter N = 16;

genvar i, j;
wire [N:0] wCarry_in [N-2:0];
wire [N:0] resultados_entrada [N-1:0];

assign resultados_entrada [0][0] = 'b0;
generate
    for (i = 0; i < N-1 ; i = i + 1 ) begin
        assign wCarry_in [i][N] = 'b0;
        assign resultados_entrada [i+1][0] = wCarry_in [i][0];
        for (j = 0; j < N ; j = j + 1 ) begin
            assign resultados_entrada[0][j+1] = a[(N-1) - j] & b[0];
            sumador mult(
            .wA( a[(N-1) - j] & b[i + 1] ) ,
            .wB(resultados_entrada [i][j]),
            .in_carry( wCarry_in [i][j+1] ),
            .wResult( resultados_entrada [i + 1][j+1] ),
            .wCarry( wCarry_in [i][j] )   );
            end
        end
endgenerate
```

Vemos que dicho código hace un barrido entre filas y columnas realizando múltiples instancias del módulo sumador de 1 bit. La lógica utilizada es complicada pero se logró el cometido, además de que se realizó parametrizado por lo que si se quiere realizar un multiplicador de 13 bits se puede cambiar el parámetro N a este valor. Para poner en la salida del módulo el resultado se utilizaron dos for, a continuación se presentará dicho código:

```
integer f,c;
always @(*) begin
    out_byte[0] = a[0] & b[0];
    for(f = 0; f < N-1; f = f + 1) begin
        out_byte[f + 1] = resultados_entrada[f+1][N];
    end
    for(c = 0; c < N; c = c + 1) begin
        out_byte[N + c] = resultados_entrada[N-1][N-1-c];
    end
end
```

Para probar este circuito se utilizó una simulación, en dicha simulación se ingresan dos números a través del firmware, que son el número 90 y 100. El resultado esperado es 9000 y se requieren varios

bits para este resultado. En la figura 11 se presenta la respuesta a dicha simulación, vemos que el número 9000 es calculado correctamente en out_byte, dicha salida es de 32 bits.

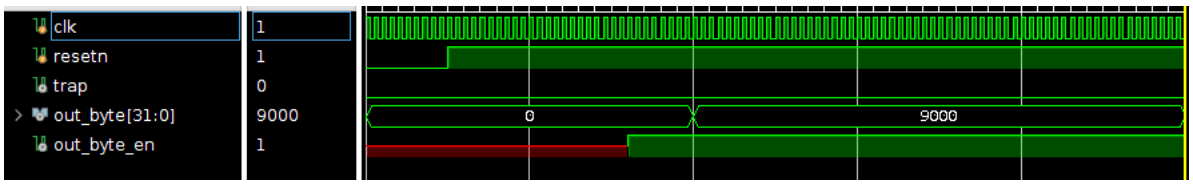

Figura 11: Multiplicación de dos números de 16 bits (90 y 100). Ejercicio 3

Este diseño generó 3544 Slice de lógica combinacional y 572 de lógica secuencial. En la figura 11 se presentan estos datos.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	3544	0	63400	5.59
LUT as Logic	1448	0	63400	2.28
LUT as Memory	2096	0	19000	11.03
LUT as Distributed RAM	2096	0		
LUT as Shift Register	0	0		
Slice Registers	572	0	126800	0.45
Register as Flip Flop	572	0	126800	0.45
Register as Latch	0	0	126800	0.00
F7 Muxes	1088	0	31700	3.43
F8 Muxes	529	0	15850	3.34

Figura 12: Cantidad de Slices de lógica combinacional y secuencial de ejercicio 3.

Al comparar la cantidad de slices del ejercicio 3 con el ejercicio 2 vemos que se aumentó casi en 400 la cantidad de slices de lógica combinaciona, esto es congruente con lo esperado ya que se realizaron 240 instancias del módulo sumador (ejercicio 3) en lugar de 12 del ejercicio 2. La cantidad de FFs se mantuvo casi constante para ambos casos, excepto por algunos agregados para una lógica de salidas, esto tiene sentido ya que los sumadores son solo de lógica combinacional.

Ahora se contestarán algunas preguntas que se presentan en la guía con respecto a este ejercicio.

3.3.1. Son los bloques generate sintetizables?.

Los bloques generate son sintetizables debido a su naturaleza, este bloque es utilizado para realizar múltiples instancias de módulos y así ahorrar tiempo a los programadores. No tendría sentido que no fuera sintetizable ya que no se usa en simulaciones.

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

El circuito realizado en la práctica 2 tenía 12 instancias de módulos sumadores, esto es porque eran dos entradas de 4 bits las que se multiplican. En este caso son entradas de 16 bits por lo que la cantidad de módulos sumadores instanciados es de 16*15 = 240. Aquí radica el poder del bloque generate ya que instanciar esto a mano sería una locura.

Con respecto al periodo de reloj, entre más etapas se añadan de lógica combinacional más va a durar el circuito en presentar la salida válida (por el tiempo de propagación entre lógica combinacional), por lo que el periodo de reloj sube y la frecuencia máxima baja. Esto produce circuitos más lentos.

3.3.3. ¿Qué ocurre con la frecuencia del circuito si se añaden latches entre cada etapa de sumadores?

Lo ideal sería utilizar Flip flops entre etapas con el fin de aumentar la frecuencia del circuito, pero esto provocaría un aumento en el tiempo en el que el resultado de la multiplicación se calcula. Los latch no son muy deseados en ningún diseño, pero en caso de agregar latchs entre las etapas no se ayudaría en mucho.

3.4. Ejercicio 4

Para este ejercicio se comienza por hacer un módulo de mux que funcione como LUT para la multiplicación de dos bits de B por A, este se muestra a continuación:

```
module LUTmux(
    input [3:0] A,
    input [1:0] B,
    input reset_L,
    output reg [7:0] mul);

    always @(*) begin
        if (reset_L) begin
            if(B == 'b00) begin
                mul = 0;
            end
            if(B == 'b01) begin
                mul = A;
            end
            if(B == 'b10) begin
                mul = A<<1;
            end
            if(B == 'b11) begin
                mul = (A<<1) + A;
            end
        end
        else begin
            mul = 'b0;
        end
    end
endmodule
```

Con este si se tiene que B es 0, la salida es cero, si B es 1, la salida es A, si B es 2, la salida es A desplazado a la izquierda, lo cual es equivalente a multiplicar por dos, y si B es 3 la salida es A

desplazado a la izquierda sumado más A. Para el multiplicador de 4bits x 4bits se tiene la siguiente implementación:

```
wire [7:0] mul1, mul2;

LUTmux mux1(A[3:0], B[1:0], resetn, mul1);
LUTmux mux2(A[3:0], B[3:2], resetn, mul2);

    always @(*) begin
temp = mul2 << 2;
        if (resetn) begin
            out_byte = mul1 + (mul2 << 2);
        end else begin
            out_byte = 0;
        end
    end
```

De esta forma se suma el resultado de la multiplicación de los dos primeros bits de B, por A más los dos siguientes bits de B por A desplazado dos bits a la izquierda, de forma que se tiene la multiplicación total de los dos números, a continuación se muestra una simulación de esto:

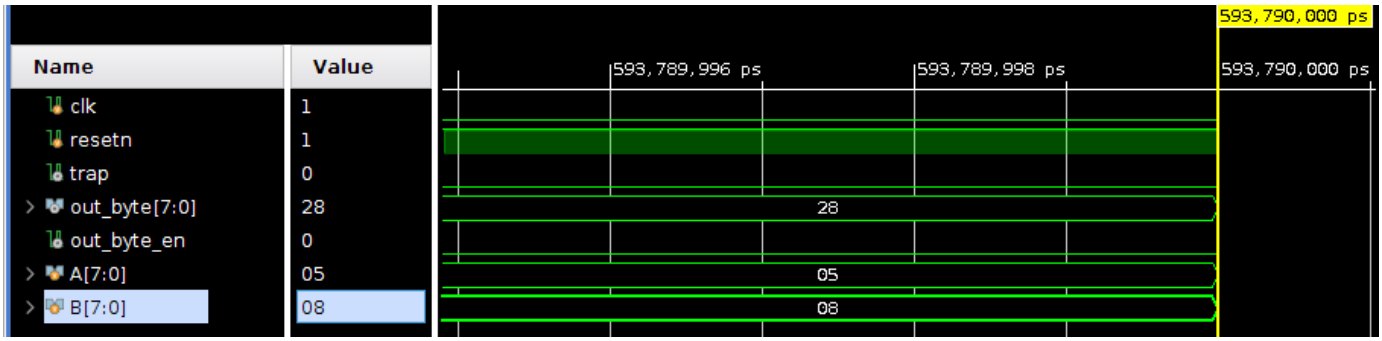

Figura 13: Simulación de multiplicación 4x4 usando LUT. Ejercicio 4

Como se puede ver se ingresa un 5 en A y un 8 en B, por lo cual el resultado debería ser 40, lo cual se obtiene en el registro out_byte como un 28 en hexadecimal, por último, se tiene el reporte de resultado de la síntesis de este código:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	3502	0	63400	5.52
LUT as Logic	1406	0	63400	2.22
LUT as Memory	2096	0	19000	11.03
LUT as Distributed RAM	2096	0		
LUT as Shift Register	0	0		
Slice Registers	827	0	126800	0.65
Register as Flip Flop	827	0	126800	0.65
Register as Latch	0	0	126800	0.00
F7 Muxes	1088	0	31700	3.43
F8 Muxes	544	0	15850	3.43

Figura 14: Reporte del resultado de la síntesis del multiplicador 4x4 con LUT. Ejercicio 4

Continuando, se vuelve a usar el módulo LUTmux, pero esta vez para el cálculo de la multiplicación de 16x16:

```
wire [31:0] mul1, mul2, mul3, mul4, mul5, mul6, mul7, mul8;

    LUTmux mux1(A[15:0], B[1:0], resetn, mul1);
    LUTmux mux2(A[15:0], B[3:2], resetn, mul2);
    LUTmux mux3(A[15:0], B[5:4], resetn, mul3);
    LUTmux mux4(A[15:0], B[7:6], resetn, mul4);
    LUTmux mux5(A[15:0], B[9:8], resetn, mul5);
    LUTmux mux6(A[15:0], B[11:10], resetn, mul6);
    LUTmux mux7(A[15:0], B[13:12], resetn, mul7);
    LUTmux mux8(A[15:0], B[15:14], resetn, mul8);

    always @(*) begin
        if (resetn) begin
            out_byte = mul1 + (mul2<<2) + (mul3<<4) + (mul4<<6) + (mul5<<8) +
            (mul6<<10) + (mul7<<12) + (mul8<<14);
        end else begin
            out_byte = 0;
        end
    end
```

Se puede apreciar que la lógica es la misma que se aplicó con anterioridad, donde se tiene el resultado de la multiplicación de dos bits de B por A en cada uno de los registros mulX, y después el se suma el resultado de estos, desplazados 2xX, donde X es la posición de los dos bits de B que se usaron para la multiplicación. A continuación se tiene la simulación de este código:

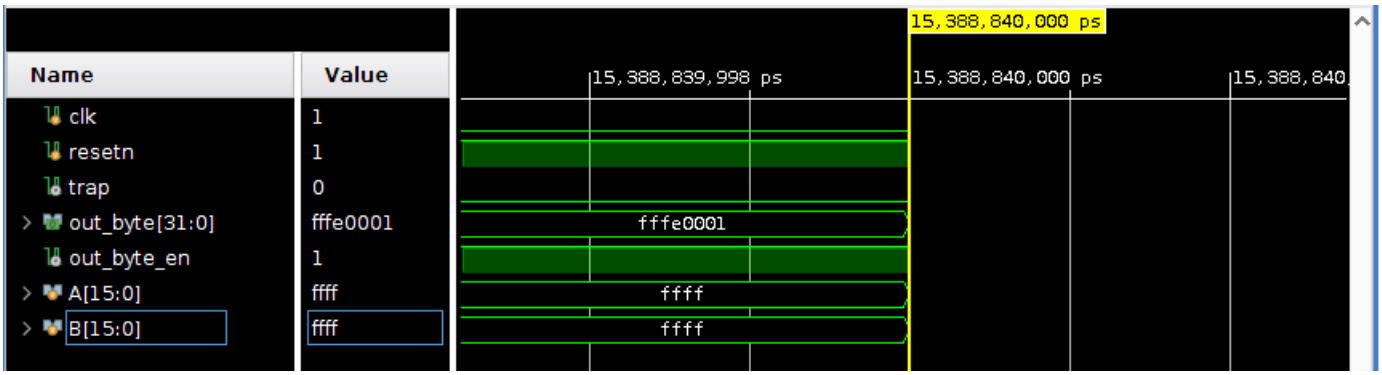

Figura 15: Simulación del multiplicador 16x16 usando LUT. Ejercicio 4

Como se puede ver, en este caso se multiplicó 0xFFFF por 0xFFFF, cuyo resultado es el de 0xFFFE0001, el cual se puede ver que se obtiene en el registro de salida, out_byte, a continuación se presenta el reporte obtenido del resultado de la síntesis:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	3838	0	63400	6.05
LUT as Logic	1742	0	63400	2.75
LUT as Memory	2096	0	19000	11.03
LUT as Distributed RAM	2096	0		
LUT as Shift Register	0	0		
Slice Registers	851	0	126800	0.67
Register as Flip Flop	851	0	126800	0.67
Register as Latch	0	0	126800	0.00
F7 Muxes	1088	0	31700	3.43
F8 Muxes	544	0	15850	3.43

Figura 16: Reporte del resultado de la síntesis del multiplicador 4x4 con LUT. Ejercicio 4

Haciendo una comparación con el multiplicador 16bitsx16bits hecho en el ejercicio 3 se puede ver que la cantidad de slices tanto de lógica combinacional como de lógica secuencial aumenta considerablemente, requiriendose alrededor de 300 slice LUTs más y 250 slice registers más.

3.5. Ejercicio 5

En este ejercicio se debe realizar un circuito que calcule el factorial de cualquier número. Este circuito varía la cantidad de ciclos de reloj que tarda en presentar el resultado del factorial dependiendo del tamaño del número, la duración en ciclos es directamente proporcional al número del factorial a calcular. Este circuito debe tomar el número desde el firmware e interceptarlo. Este circuito comienza el cálculo del factorial cuando se pone en 1 el valor de memoria de la dirección 0x0FFFFFF4. El resultado es guardado en 0x0FFFFFF8 y se pone un 1 en la dirección 0x0FFFFFFC cuando el resultado está listo. El circuito de la implementación se presenta a continuación:

```
reg [31:0] in;
wire [31:0] out;
reg [15:0] result_aux;
reg [15:0] contador;

always @(posedge clk) begin
    if (b == 'b1) begin
        if (mem_la_write && mem_la_addr == 32'h0FFF_FFF0) begin
            contador <= mem_la_wdata - 1;
            result_aux <= mem_la_wdata;
        end
        else  begin
            if(contador != 'b1) begin
                contador <= contador - 1;
                result_aux <= out;
            end
        end
    end
end

multiplicador mulws(
    .a (contador),
    .b (result_aux),
    .out_byte_a (out)
);
```

En esta implementación lo que se hace es utilizar el módulo multiplicador de 16 bits del ejercicio 3, se realiza la instancia del mismo y se toman como entradas una variable contador y el result_aux, la salida es llamada out. Lo que hace este código es tomar el número de entrada (en el ejemplo de la simulación 5) y se multiplica por (5-1), este resultado es retroalimentado a la entrada y se multiplica por (5-2), esta operación se repite hasta que se llegue a (5-4) y es cuando está listo el factorial. Esto se presenta cuando la variable contador llega a 1.

En la figura 18 vemos el cálculo del factorial de 5 (in = 5), el resultado esperado es 120 en decimal y en hexadecimal es 78 (out_byte = 78), vemos que el resultado es correcto. En dicha figura también podemos ver que después de terminar el cálculo del factorial se accede a la dirección de memoria 0x0FFFFFF8, esto es porque se coloca el resultado del factorial en esta posición. También se accesa la dirección 0x0FFFFFFC y esto es porque se pone un 1 en esta locación para indicarle al programador que el factorial ya se terminó de calcular.

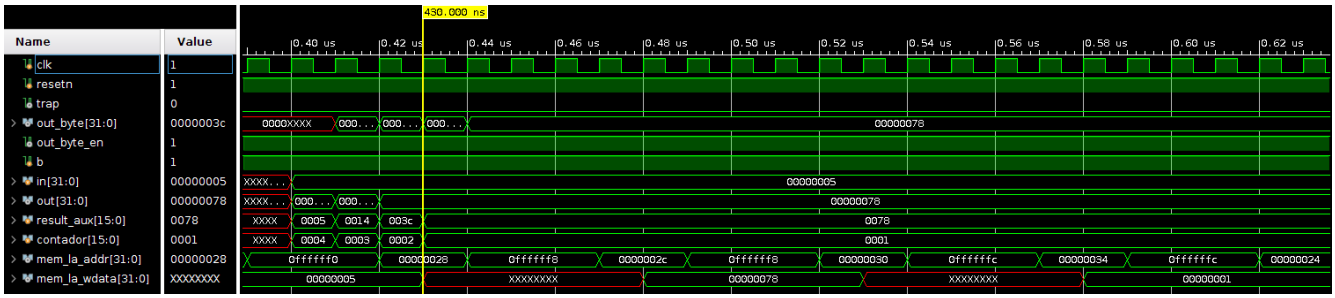

Figura 17: Cálculo del factorial, ejercicio 5.

En la figura 18 podemos ver la cantidad de compuertas combinacionales y secuenciales registradas para este diseño:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	3401	0	63400	5.36
LUT as Logic	1305	0	63400	2.06
LUT as Memory	2096	0	19000	11.03
LUT as Distributed RAM	2096	0		
LUT as Shift Register	0	0		
Slice Registers	588	0	126800	0.46
Register as Flip Flop	588	0	126800	0.46
Register as Latch	0	0	126800	0.00
F7 Muxes	1072	0	31700	3.38
F8 Muxes	513	0	15850	3.24

Figura 18: Reporte de slices, ejercicio 5.

La cantidad de ciclos de reloj que se tarda en calcular el factorial de un número N, es N, ya que es proporcional al tamaño del mismo, debido a que las iteraciones que tiene que hacer con respecto a ciclos de reloj es N.

Haciendo una comparación entre los slices utilizados para la implementación del código de cálculo de factorial del ejercicio 1 mediante código en el firmware y utilizando la extensión RV32M y mediante el uso de uno de los multiplicadores de los ejercicios anteriores (en este caso se usó el multiplicador 16x16 con bloques sumadores), se tiene que la implementación del primero requiere por mucho, menos hardware, ya que el primero se implementa con poco más de un tercio de los slices de lógica combinacional con los que se implementa el segundo, sin embargo, respecto a los slices de lógica secuencial la implementación por medio del firmware usa aproximadamente 200 slices más.

4. Enlace del repositorio que contiene el código fuente, junto con el respectivo commit ID.

Enlacea repositorio de GitLab: https://gitlab.com/YeisonRP/Lab_digitales_2_B56074_B57082

5. Conclusiones y recomendaciones

A continuación se presentan las principales conclusiones encontradas en este experimento:

- Existen diversas extensiones para el picorv32 que añaden paquetes de instrucciones nuevas a la hora de ensamblar el código, las cuales pueden facilitar el trabajo con multiplicaciónes, divisiones, punto flotantes, ya sea precisión simple o precisión doble.
- El uso del LUT para el cálculo de multiplicaciones y otras operaciones resulta útil, ya que reduce un cálculo que puede ser complicado de implementar en hardware a muxes que seleccionan valores en función de las entradas de la tabla.
- Es posible realizar circuitos parametrizados en verilog para que realicen tareas específicas, como el multiplicador del ejercicio 3 que utilizando el bloque generate se puede generar un multiplicador de cualquier cantidad de bits. Además de que se ahorra una gran cantidad de tiempo de no realizar instancias repetitivas de manera manual.
- Utilizando la herramienta vivado se puede saber la cantidad de compuertas combinacionales y secuenciales que generó el circuito, además de la frecuencia de reloj máxima, entre otras características de la implementación.
- Se puede calcular un número factorial con el procesador RISC-V (como es lo usual en una computadora normal) ya que la implementación utilizada permite activar instrucciones de multiplicación, o también generar un circuito combinando lógica secuencial y combinacional para calcular el factorial de cualquier número en varios ciclos de reloj.
- Existen diversas formas de implementar circuitos en hardware para realizar multiplicaciones, algunos son más eficientes que otros en cantidad de compuertas.
- Se debe tener cuidado al crear circuitos de lógica combinacional que realicen grandes cálculos, ya que podrían presentar una disminución en la frecuencia de reloj en el caso de la FPGA, o en un circuito real podrían darse problemas en tiempo de setup y hold.

Seguidamente se mencionarán algunas recomendaciones para facilitar la recreación de este experimento.

- Al usar shift left o shift right en el código de verilog, es preferible encerrar el desplazamiento en un paréntesis, esto ya que de otra forma puede estar considerando variables extra en el cálculo de los bits a desplazar, las cuales puede no desearse.
- Al desplazar un registro, aún cuando este no sea el registro destino, se debe verificar que se cuenta con la cantidad de bits disponibles necesarios en este registro, para de esta forma no perder datos en el desplazamiento.
- Estudiar de manera detallada el bloque generate de verilog, ya que se debe tener claro el concepto del mismo para poder realizar arreglos complicados, como lo es el multiplicador del ejercicio número 3. Además de realizar diagramas del circuito a generar y pensar en forma matricial con respecto a los cables.

- Escribir los códigos del firmware y el archivo de constraints antes de realizar la práctica de laboratorio, debido a que se tendrá un mayor entendimiento del problema a resolver, además de que se ahorrará tiempo que será necesario para terminar la práctica a tiempo. También se debe realizar diagramas de flujo de los circuitos de verilog a diseñar.
- Aprender a buscar y entender los reportes generados por vivado con respecto a la cantidad de compuertas combinacionales y secuenciales, con el fin de comparar los diseños que se realizan en esta práctica y entender cuál fue más eficiente.