Universidad de Costa Rica

Bitácora

Experimento 1

Prof. José Daniel Hernández

Luis Soto Camacho, B57082
Yeison Rodríguez Pacheco, B56074

Grupo 2

30 de agosto de 2019

Índice

Índice de figuras

Índice de tablas

1. Objetivos

1.1. Objetivo general

- Introducir las herramientas que serán utilizadas durante el curso mediante una práctica introductoria sobre la síntesis de código HDL en un FPGA.

1.2. Objetivos específicos

- Familiarizar al estudiante con el ambiente integrado de desarrollo Vivado de Xilinx.
- Presentar al estudiante una implementación en HDL de un microprocesador de la arquitectura RISC-V.
- Presentar al estudiante un ejemplo sencillo en código C para ser ejecutado en el microprocesador.

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

Describa la interacción que existe entre el firmware, el procesador Picorv32 y los LEDs. ¿Cómo inluye el firmware en la cuenta que se despliega en los LEDs? ¿Cómo está compuesto el espacio de memoria que ve el procesador Picorv32?

Comencemos por analizar el firmware, este es un código en lenguaje C que se encarga de de contar desde 0 hasta 255 y poner cada número de esta cuenta en la dirección de memoria $0x10000000$. Entre cada cuenta se espera un tiempo con la ayuda de un while que repite la misma iteración 1000000 veces (esto es para realizar un delay).

Este firmware se compila para la arquitectura RISC-V, que es la arquitectura del procesador Picorv32. Este archivo compilado es guardado en una memoria y cuando el circuito empieza a funcionar, el procesador comienza a leer esta memoria por lo que ejecuta el firmware, que es el código explicado con anterioridad.

El código de verilog del archivo system.v tiene un segmento vital que es el siguiente:

```
if (mem_la_write && mem_la_addr == 32'h1000_0000) begin
    out_byte_en <= 1;
    out_byte <= mem_la_wdata;
end
```

En este código podemos ver que cada vez que el procesador tenga en su bus de direcciones (en el siguiente ciclo) la dirección de memoria 0x10000000 y se levante la bandera de escritura (en lookahead) se pone en la variable out_byte el número que leyó el procesador. Este número es producido por el firmware explicado anteriormente y es una cuenta de 0 a 255. Esta salida out_byte se conecta por medio del archivo de constraints a los leds y es por esto que se enciendn haciendo una cuenta de 0 a 255.

3.2. Ejercicio 2

Puede notar que el número que se despliega en los LEDs cambia muy lentamente. Modifique el código en la para que este cambio sea más rápido.

La modificación del código se realizó en el firmware, ya que se consideró que era la forma más sencilla de resolver el problema. Simplemente se dividió la constante limit a la mitad para que el tiempo entre cambios sea la mitad. Esto es porque dicha constante se utiliza en un while para esperar una cierta cantidad de iteraciones. A continuación se presenta el código modificado:

```
#include <stdint.h>

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

void main() {
    uint32_t number_to_display = 0;
    uint32_t counter = 0;
    const uint32_t limit = 500000;

    while (1) {
         counter = 0;
         putuint(number_to_display);
         number_to_display++;
         while (counter < limit) {
             counter++;
         }
    }
}
```

A continuación se muestra una simulación de este mismo código, pero disminuyendo su conteo 100000 veces, de forma que se pudiera ilustrar bien en el tiempo de simulación:

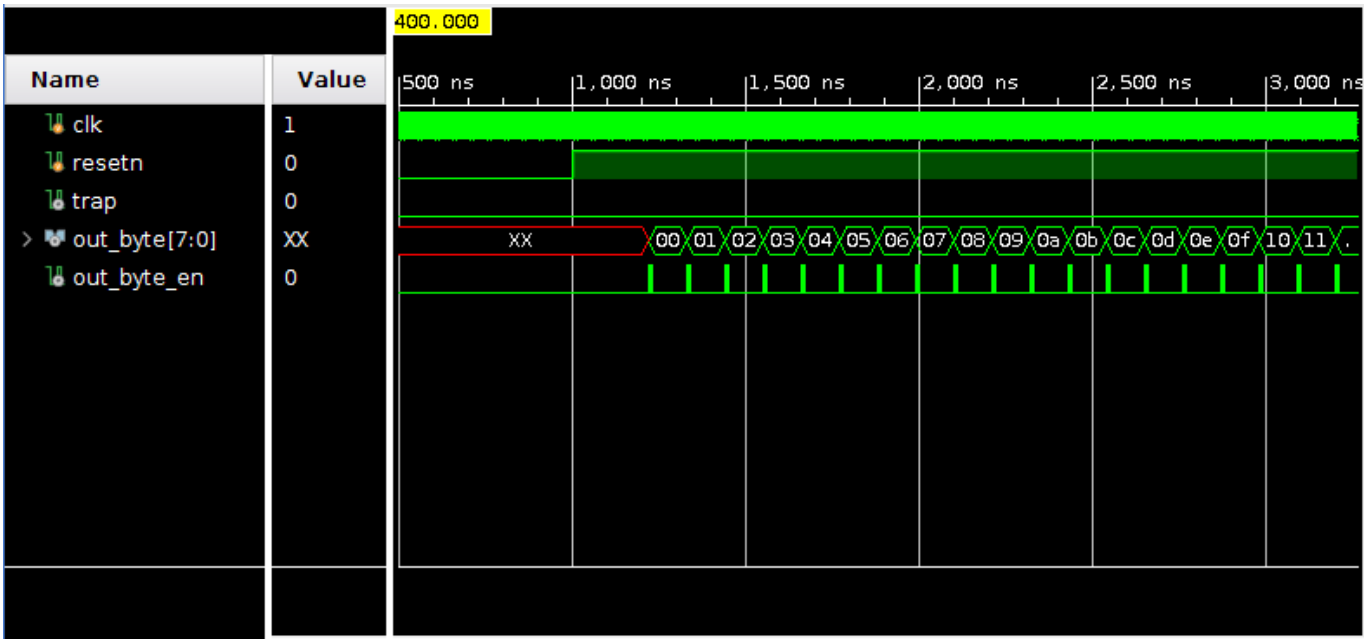

Figura 1: Simulación del código de conteo

3.3. Ejercicio 3

Realice los cambios necesarios para desplegar el contador utilizando el display de 7 segmentos de la tarjeta de desarrollo. Para esto deberá de cambiar el archivo de constraints. Investigue para qué sirve el archivo de constraints y cómo se puede modificareste archivo desde la interfaz gráfica. Realice los cambios necesarios.

Para utilizar el display de 7 segmentos se realizó un cambio en el archivo de verilog *system.v*. A continuación se presenta la modificación:

```
if (contador1 < 512) begin
    case (out_byte_aux[3:0])
    4'b0000  : out_byte <= 'b00000011;
    4'b0001  : out_byte <= 'b10011111;
    4'b0010  : out_byte <= 'b00100101;
    4'b0011  : out_byte <= 'b00001101;
    4'b0100  : out_byte <= 'b10011001;
    4'b0101  : out_byte <= 'b01001001;
    4'b0110  : out_byte <= 'b01000001;
    4'b0111  : out_byte <= 'b00011111;
    4'b1000  : out_byte <= 'b00000001;
    4'b1001  : out_byte <= 'b00001001;
    4'b1010  : out_byte <= 'b00010001;
    4'b1011  : out_byte <= 'b11000001;
    4'b1100  : out_byte <= 'b01100011;
    4'b1101  : out_byte <= 'b10000101;
    4'b1110  : out_byte <= 'b01100001;
    4'b1111  : out_byte <= 'b01110001;
    default : out_byte <= 'b00010000;
endcase
end
else begin
    case (out_byte_aux[7:4])
     4'b0000  : out_byte <= 'b00000011;
     4'b0001  : out_byte <= 'b10011111;
     4'b0010  : out_byte <= 'b00100101;
     4'b0011  : out_byte <= 'b00001101;
     4'b0100  : out_byte <= 'b10011001;
     4'b0101  : out_byte <= 'b01001001;
     4'b0110  : out_byte <= 'b01000001;
     4'b0111  : out_byte <= 'b00011111;
     4'b1000  : out_byte <= 'b00000001;
     4'b1001  : out_byte <= 'b00001001;
     4'b1010  : out_byte <= 'b00010001;
     4'b1011  : out_byte <= 'b11000001;
            4'b1100  : out_byte <= 'b01100011;
```

```
      4'b1101  : out_byte <= 'b10000101;
      4'b1110  : out_byte <= 'b01100001;
      4'b1111  : out_byte <= 'b01110001;
      default : out_byte <= 'b00010000;
endcase
end
```

Lo que se hizo fue tomar el byte de entrada que encendía los leds anteriormente, y se dividió en los primero 4 bits y los segundos 4 bits. Con esto se logra que los primeros 4 bits se muestren en el primer display y los otros 4 en el segundo.

Pero esto no es tan sencillo ya que solo se puede utilizar un display al mismo tiempo, por lo que se realizó un contador que cada 512 ciclos de reloj cambia de un display al otro, y por ende también cambia el número a la salida (out_byte). Este cambio ocurre tan rápido que el ojo humano no nota que en realidad solo 1 display está encendido al mismo tiempo, por lo que se provoca el efecto de que se vean dos números en los display al mismo tiempo.

Como ejemplo para dejar más claro el funcionamiento, si el procesador lee el número en binario 0001 0010, el primer display en los primeros 512 ciclos de reloj tiene la codificación del número 2 en siete segmentos (00100101), en los siguientes 512 ciclos de reloj se cambia la codificación al número uno (10011111) y se pone a funcionar el display número dos. Esto hace que el los display de 7 segmentos muestren el número 12. La codificación fue realizada en hexadecimal por lo que la cuenta llega hasta FF.

Desde la GUI de vivado se puede abrir el archivo de constraints que tiene extención *.xdc* y modificarse, esto con el fin de conectar a los display de 7 segmentos la salida out_byte. En la imagen 5 se presenta el archivo modificado de constraints desde la GUI.

system.xdc

/home/yeison/inicio-ie424/src/constraints/system.xdc

```
1  set_property CFGBVS VCCO [current_design]
2  set_property CONFIG_VOLTAGE 3.3 [current_design]
3
4  # clk
5  set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
6  create_clock -add -name sys_clk_pin -period 10.00 [get_ports {clk}];
7
8  # switches
9  set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { resetn }];
10
11 # leds
12 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { out_byte[7] }];
13 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { out_byte[6] }];
14 set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { out_byte[5] }];
15 set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { out_byte[4] }];
16 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { out_byte[3] }];
17 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { out_byte[2] }];
18 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { out_byte[1] }];
19 set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { out_byte[0] }];
20
21 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[0] }];
22 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[1] }];
23 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[2] }];
24 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[3] }];
25 set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[4] }];
26 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[5] }];
27 set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[6] }];
28 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { out_enable_digit[7] }];
29
30 set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { out_byte_en }];
31 set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { trap }];
32
```

Figura 2: Archivo de constraints modificado para encender led de 7 segmentos.

El archivo de constraints es utilizado por vivado para realizar las conexiones entre las entradas y salidas del RTL con los pines reales del circuito, en este caso se encuentra escrito en el archivo system.v, por lo que si no se realizaran conexiones en el archivo de constraints, en realidad el circuito del RTL ni si quiera estaría conectado, este también permite hacer una interpretación de los pines del circuito, de forma que se le puede asignar un nombre personificado a cada uno de los pines, esto resulta muy útil a la hora de trabajar, ya que facilita la abstracción del funcionamiendo del dispositivo y hace que sea más consistente con el diseño, y evita el uso de los nombres genéricos que poseen los pines originalmente.

En la figura 3 se presenta la simulación del display de 7 segmentos con el fin de demostrar su funcionamiento. Vemos que en el mismo ciclo que se levanta la señal output_byte_enable se cambia el número en la salida output_byte. Esta salida en los primeros ciclos es la codificación de un 0 en el display de 7 segmentos (03), luego sigue la codificación del 1 (9f) y así sucesivamente, lo que provoca que se enciendan los leds del display. Es importante recalcar que para la simulación se cambió el firmware para que el tiempo de espera fuera mucho más corto debido a que las simulaciones tienen tiempo exponencial y tardan mucho en ejecutarse.

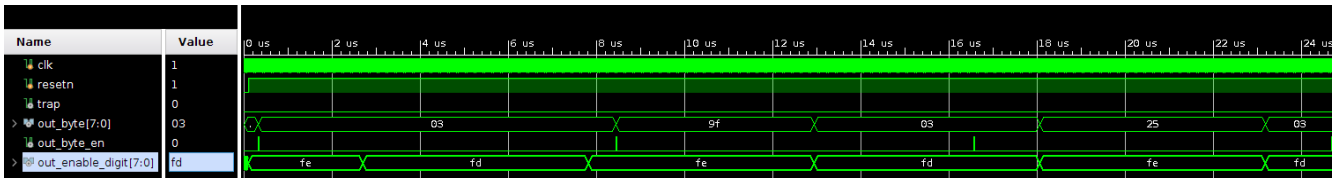

Figura 3: Simulación del circuito que enciende el display de 7 segmentos

3.4. Ejercicio 4

Utilizando una simulación demuestre que la implementación del procesador sigueun diseño de pipelining. Identifique las etapas del pipeline y demuestre para lasprimeras 10 instrucciones cuáles etapas se llegan a ejecutar. Justifique losresultados obtenidos

Para la demostración de la existencia del pipeline en el procesador se debe analizar el comportamiento del registro de pc, para esto se hace uso del siguiente código:

```
uint32_t number_to_display = 0;
uint32_t counter = 0;
const uint32_t limit = 1000000*8;

while (1) {
     counter = 0;
     putuint(number_to_display);
     number_to_display++;
     while (counter < limit) {
        counter++;
     }
}
```

Simulando este mismo, se obtienen la siguiente gráfica:

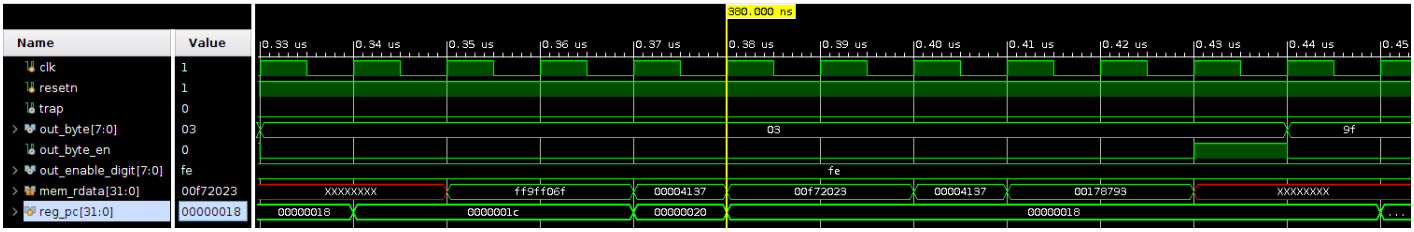

Figura 4: Simulación para observar el comportamiento del registro PC, con el fin de demostrar la inexistencia del pipeline

En esta, se puede apreciar que el valor almacenado en el registro de PC avanza por el código como es de esperarse, sin embargo, este tarda más en varios de las instrucciones que se están ejecutando, ya que como se puede ver, para la instrucción en 0x0000001C se tiene que el procesador tarda 3 ciclos de reloj ejecutandola, mientras que para la instrucción en 0x00000020 se tarda 1 solo ciclo de reloj en ejecución y para la almacenada en 0x00000018 se tardan 7 ciclos de reloj. En un procesador con pipeline, se tiene que el contador de pc cambia con cada ciclo de reloj, ya que con cada ciclo nuevo, los datos de la etapa anterior avanzan a la siguiente, y se carga una nueva instrucción, pero este comportamiento no se está cumpliendo en este procesador, de forma que este espera a que la instrucción se complete y pase por todo el circuito combinacional, por lo que es correcto afirmar que en este procesador no se usa pipeline, lo cual a su vez significa que no posee etapas.

Utilizando una simulación identifique un caso de read after write. Indique si el caso se maneja correctamente y si tiene algún efecto en el pipeline. Describa cómo se maneja el caso

Buscando obtener una situación donde se escriba un dato y justamente después se lea este, se programó el siguiente código en el archivo de firmware:

```
uint32_t number_to_display = 0;
uint32_t counter = 0;
uint32_t asd = 3;
const uint32_t limit = 1000000*8;

while (1) {
    counter = 0;
    putuint(number_to_display);
    number_to_display++;
    while (counter < limit) {
        counter++;
        asd = counter;
        asd++;
    }
}
```

En este código, se consigue escribir el dato counter, el cual después de esto se procede a leer, para igualar el dato *asd* a este, estas dos instrucciones se ven de la siguiente forma simuladas:

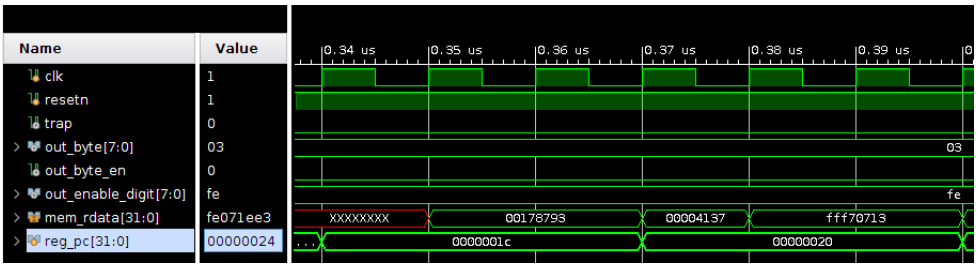

Figura 5: Simulación de read after write.

Al observar el opcode de las instrucciones a ejecutar en el registro mem_rdata vemos que la primera es un addi y la segunda un lui, por lo que es en este instante cuando se da el caso read after write. Como se puede ver, no hay problema al ejecutar la instrucción, esto se debe a que al no poseer etapas ni funcionamiento de pipeline, el riesgo de read after write no tiene relevancia, ya que el circuito espera hasta que se escriba el dato, y después de esto carga la instrucción que la lee, por lo que nunca se tiene un problema con la integridad de la información.

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:

- Es posible utilizar una implementación de un procesador arquitctura RISC-V en la FPGA Nexys 4 DDR, cargar firmware al mismo (escrito en C) por medio de una memoria (escrita en verilog) y realizar implementaciones extras utilizando el procesador como cerebro en las operaciones.
- Se demostró la utilidad de las simulaciones comprobando que el procesador PICORV32 no tiene diseño de pipeline, lo cuál hubiese sido más complicado si solo se pudiesen realizar implementaciones en la tarjeta, debido a los rápidos tiempos de ejecución.
- Se comprobó que el archivo de constraints efectivamente maneja las conexiones entre los pines de entrada y salida del módulo top de verilog con los pines de la tarjeta Nexys 4 DDR.
- Al navegar por la interfaz de vivado se encontró que tiene múltiples tanto de visualización como de análisis a la hora de ejecutar todo el flujo de diseño de un código escrito en RTL, como lo son DRC, que el código sea sintetizable, cantidad de compuertas combinacionales y secuenciales del diseño, reportes de potencia, etc.

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

- Estudiar de la página de Xilinx la sintaxis que se utiliza en el archivo de constraints. También leer la documentación del repositorio del procesador PicoRV32.

- Aprender a utilizar la herramienta vivado, ya que es un software poderoso pero en un comienzo puede ser algo confuso. En línea se encuentran múltiples tutoriales sobre su uso.
- Tener conocimiento sobre programación en Verilog, además de tener clara la diferencia entre un testbench y un diseño sintetizable, ya que la herramienta posee un simulador y un sintetizador por lo que esto podría generar confusión si no se tienen claros estos conceptos.
- Estudiar y tener acceso al diagrama de pines de la FPGA que se esté utilizando, así como su documentación para realizar las consultas respectivas de cómo funcionan ciertas secciones de la FPGA.
- Tener claros los conceptos de pipelining y el funcionamiento general de las etapas de un procesador arquitectura RISC-V. Además de contar con el set de instrucciones y la codificación de las mismas.
- Ser consciente de que el diseño que se haga en RTL va a ser una implementación física, por lo que los tiempos en que ocurran los cambios deben ser considerados. Por ejemplo al utilizar el display de 7 segmentos, si los cambios entre ambos display son muy rápidos, los leds no se terminarán de apagar por lo que resulta en una visión difusa de los números en la tarjeta.