Tecnológico de Monterrey

Campus Monterrey

Materia

Arquitectura de computadoras (Gpo 1)

Tema

Práctica #3. Memoria de instrucciones y memoria de datos

Integrantes

- ➔ Jorge Besnier A01039882
- ➔ Cinthia Portillo A00811827

Maestro

Diego Valencia

Fecha
10/09/20

Introducción

En esta ocasión se desarrollaron dos módulos de memoria para la arquitectura de computadoras. En primera instancia la memoria de instrucciones de un procesador que es solamente de lectura ya que esta almacena en 1's y 0's las operaciones que puede hacer una computadora, mejor conocida como memoria ROM. Por otra parte, se desarrolla una memoria de datos, o mejor conocida como RAM, que tiene la capacidad de escritura y lectura de palabras.

A continuación se presenta el contenido del reporte.

Contenido

Introducción **2**

Contenido **2**

Memoria de instrucciones **3**

Resultados de TestBench Memoria de instrucciones 4

Memoria de datos **5**

Direccionamiento de las memorias 6

Desarollo de Memoria RAM 7

Código Test Bench de Memoria Ram 8

Resultados de TestBench 10

Conclusión **11**

Cinthia 11

Jorge 11

Bibliografía **11**

Memoria de instrucciones

Como se mencionó previamente, la memoria de instrucciones, se implementa utilizando una memoria tipo ROM (Read Only Memory), es la memoria donde se almacenan las instrucciones del programa que se deben ejecutar.
A continuación se mostrará el desarrollo de un módulo VHDL para la memoria de instrucciones que cumple con las siguientes características.

- Contar con 32 espacios de memoria donde cada uno alberga palabras de 32 bits
- Tener una entrada READ_ADDRESS[31:0] y una salida INSTRUCTION[31:0] . De la salida INSTRUCTION[31:0] se obtiene la instrucción (el conjunto de 1's y 0's según el formato de la arquitectura MIPS) que se encuentra en la dirección READ_ADDRESS[31:0] . En otras palabras, la variable READ_ADDRESS se utiliza como el índice de la dirección a leer.

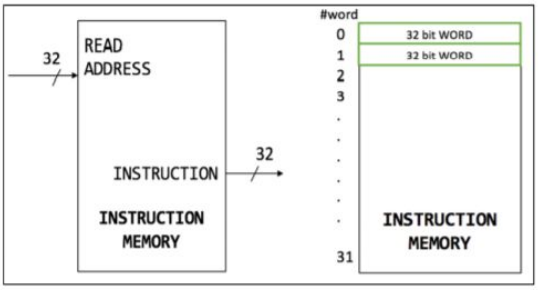

Fig.1.1Memoria de instrucciones con palabras de 32 bit

```
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5  use IEEE.NUMERIC_STD.ALL;
6  use IEEE.STD_LOGIC_UNSIGNED.ALL;
7
8  -- Uncomment the following library declaration if using
9  -- arithmetic functions with Signed or Unsigned values
10 --use IEEE.NUMERIC_STD.ALL;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx primitives in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17 entity instructionMem is
18 port(
19       READ_ADDRESS: in std_logic_vector(31 downto 0);
20       INSTRUCTION: out std_logic_vector(31 downto 0)
21       );
22
23 end instructionMem;
24
25 architecture Behavioral of instructionMem is
26
27 type ROM is array(0 to 31) of std_logic_vector(31 downto 0);  --crando memoria 32 datos entrada x 32 de salida,
28 constant ROM_MEMORY: ROM := ( x"00000000",  -- contenido de la direccion 0
29                                x"00000001",  -- contenido de la direccion 4
30                                x"00000002",  -- contenido de la direccion 8
31                                x"00000003",  -- contenido de la direccion C
32                                others => (others => '0'));
33
34 signal DIV: UNSIGNED(31 downto 0);
35 signal DIV_ADDRESS: UNSIGNED(31 downto 0);
36
37 begin
38
39 DIV_ADDRESS <= UNSIGNED(READ_ADDRESS);
40 DIV <= DIV_ADDRESS / "100"; --division del address entre 4 para acceso secuencial a las localidades
41
42 INSTRUCTION <= ROM_MEMORY(to_integer(DIV));
43
44 end Behavioral;
```

Fig.1.2 Código Fuente de la Memoria de instrucciones

En el código anterior se crea una memoria de 32 datos de entrada con 32 de salida, se establece su dirección y su contenido para posteriormente en el código de test bench poder leer el contenido de 3 direcciones dadas

```
    COMPONENT instructionMem
    PORT(
         READ_ADDRESS : IN  std_logic_vector(31 downto 0);
         INSTRUCTION : OUT  std_logic_vector(31 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal READ_ADDRESS : std_logic_vector(31 downto 0) := (others => '0');

   --Outputs
   signal INSTRUCTION : std_logic_vector(31 downto 0);

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: instructionMem PORT MAP (
          READ_ADDRESS => READ_ADDRESS,
          INSTRUCTION => INSTRUCTION
        );



   stim_proc: process
   begin
      -- hold reset state for 50 ns.
      wait for 50 ns;
      -- insert stimulus here
      READ_ADDRESS <= x"00000000";
      wait for 50 ns;
      READ_ADDRESS <= x"00000004";
      wait for 50 ns;
      READ_ADDRESS <= x"0000000C";
```

Fig.1.2 Código test bench de la Memoria de instrucciones

Resultados de TestBench Memoria de instrucciones

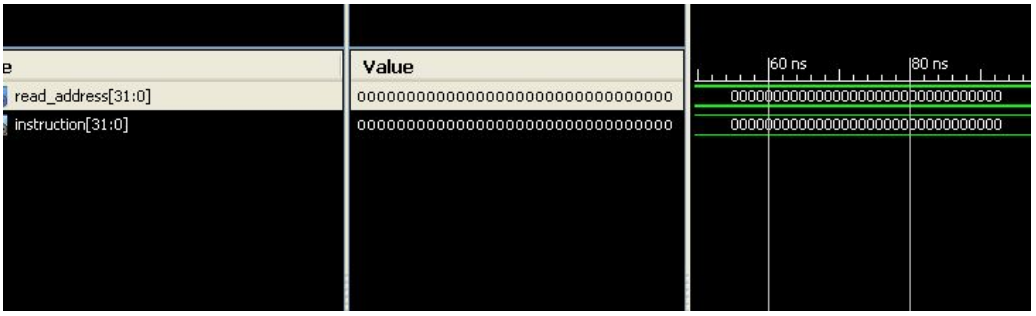

Fig.1.3 Resultado de leer la dirección 0 y nos muestra su contenido

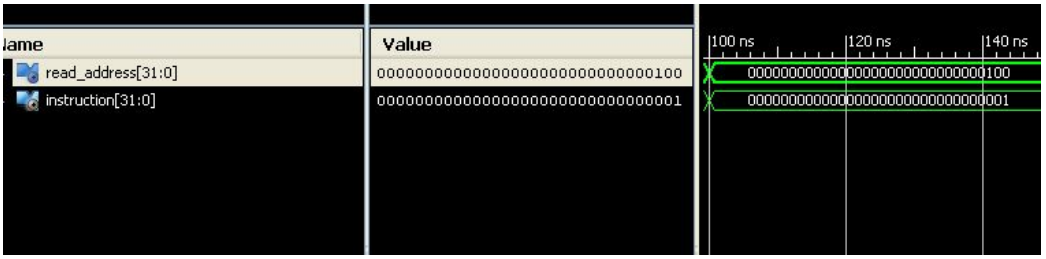

Fig.1.4 Resultado de leer la dirección 4 y nos muestra su contenido

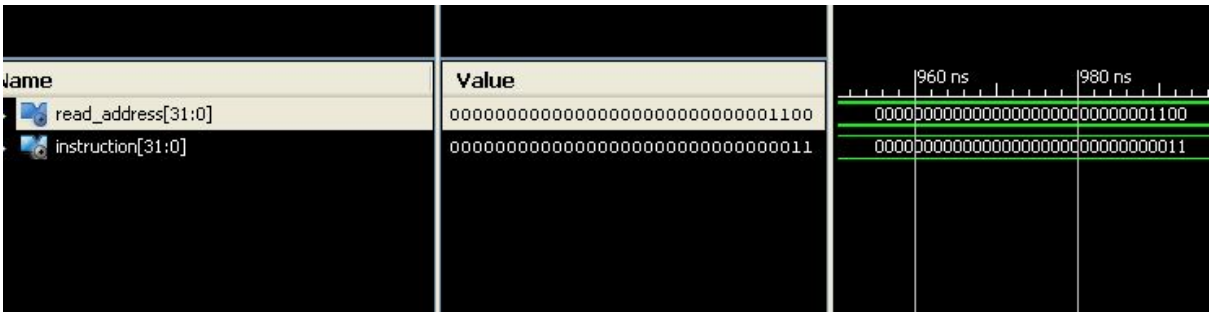

Fig.1.5 Resultado de leer la dirección 12 y nos muestra su contenido

Memoria de datos

La memoria de Datos, por lo general es una memoria tipo RAM (Random Access Memory) cuyas operaciones, ya sean de escritura o lectura de datos, son de acceso aleatorio. La 'Data Memory' contiene los datos que serán cargados a los registros o que son almacenados por alguna instrucción.

Como verás más adelante, las operaciones aritméticas y lógicas son efectuadas sobre los valores actuales de los registros del procesador y, dado que estos registros son limitados (32 para nuestro MIPS), el resto de los datos con los que trabajamos se quedan en memoria hasta ser necesitados.

La siguiente tabla contiene las entradas y salidas de la memoria.

Tabla 3.1. EE/SS de la memoria de datos	
Entradas: + ENABLE + WRITE_ENABLE + READ_ENABLE + CLK + RW_ADDRESS + WRITE_DATA	Salidas: + READ_DATA

Ahora, deberás crear un módulo VHDL distinto para el diseño de la memoria de datos que cumpla con lo siguiente.

- Las entradas de WRITE_ENABLE y READ_ENABLE son habilitadores para los procesos de lectura y escritura, nunca pueden estar los dos en '1' al mismo tiempo.
- La lectura consiste en poner en READ_DATA el contenido de memoria de la dirección especificada por ADDRESS.

- El proceso de escritura almacena en memoria en la dirección dada por ADDRESS el dato que entra a WRITE DATA cada que ocurra una transición negativa de reloj por la entrada de CLK.
- En el proceso de escritura, READ_DATA “saca” puros 0’s puesto que no leyó ningún dato.
- ENABLE resulta un habilitador general de la memoria en sí (notar que está negado). Para efectos de esta práctica, dicha entrada no será requerida sino hasta las prácticas finales.
- Nota que la lectura no está asociada a un ciclo de reloj.

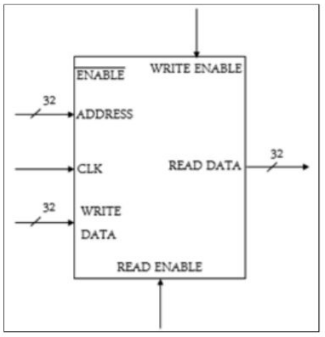

Figura 3.2. Memoria de datos

Direccionamiento de las memorias

Como ya habrás notado en la práctica que realizaste el módulo ‘program counter’, el incremento de éste va de 4 en 4. Esto es debido a que las instrucciones para 13 la arquitectura MIPS se componen de 4 bytes cada una. Y es precisamente la salida del ‘PC’ la misma señal de entrada de ADDRESS de nuestra memoria de instrucciones. A continuación, se muestra un ejemplo de cómo se puede definir una memoria dentro de un ‘process’.

```
PROCESS (Lista_sensitividad)
SUBTYPE REGISTRO IS STD_LOGIC_VECTOR (31 DOWNTO 0);
TYPE REG_BANK IS ARRAY (0 TO 31) OF REGISTRO;
VARIABLE ROM_MEMORY: REG_BANK: = (
                                    X"0000",
                                    X"0001",
                                    X"0002",
                                    X"0003",
                                     OTHERS => (OTHERS => '0')
                                );
BEGIN
   << cuerpo del process >>
END PROCESS;
```

Como ya se introdujo anteriormente, solo necesitamos 5 bits para direccionar nuestros módulos-memorias. De igual forma el índice interno de nuestras memorias, si se realiza como el ejemplo anterior, es un número natural , por lo que debemos dividir nuestro valor de ADDRESS entre 4 para acceder secuencialmente a las localidades (Recuerda que el PC aumenta de 4 en 4).

Dirección De Memoria					Número de palabra (índice de memoria interna)
0x00	BYTE 0	BYTE 1	BYTE 2	BYTE 3	palabra 1
0x04	BYTE 4	BYTE 5	BYTE 6	BYTE 7	palabra 2
0x08	BYTE 8	BYTE 9	BYTE A	BYTE B	palabra 3
0x0C	BYTE C	BYTE D	BYTE E	BYTE F	palabra 4

Figura 3.3. Acomodo de memoria en nuestro MIPS

De acuerdo con la figura anterior, podemos notar que si se quiere acceder a la segunda palabra de la memoria, la instrucción de ensamblador deberá contener la literal 0x04 , para la tercera palabra la 0x08 y así sucesivamente.

Para nuestras memorias internas sucede algo como lo siguiente:

b0000_0000_0000_0(000_01)00 => 0x04, ... 1
b0000_0000_0000_0(000_10)00 => 0x08, ... 2
b0000_0000_0000_0(000_11)00 => 0x0C, ... 3

Donde los 5 bits resaltados entre paréntesis nos dan el índice natural al que debemos acceder para que exista correspondencia entre el [PC + 4] y nuestras direcciones internas que van de 1 en 1.

Desarollo de Memoria RAM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity Mmemoria is
7      Port (
8             WRITE_ENABLE : in  STD_LOGIC;
9             READ_ENABLE : in  STD_LOGIC;
10            CLK : in  STD_LOGIC;
11            RW_ADRDRESS : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
12            WRITE_DATA : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
13            READ_DATA : out  STD_LOGIC_VECTOR(31 DOWNTO 0));
14 end Mmemoria;
15
16 architecture Behavioral of Mmemoria is
17    --determinar la posicion inicial de la memoria
18    signal POSINICIAL : STD_LOGIC_VECTOR(31 DOWNTO 0) := X"00000000";
19    --variables auxiliares para saber si ambos son 0 o ambos son 1
20    SIGNAL RAUX : STD_LOGIC;
21    SIGNAL WAUX : STD_LOGIC;
```

Fig 2.1 Definición del módulo VHDL de la memoria RAM

La variable temporal de POSTINICIAL nos facilita el direccionamiento con posición inicial diferente de cero. Por lo tanto, con este código podemos recibir cualquier dirección configurando esta variable con la posición inicial con la que vamos estar trabajando.

```
begin

   --LA ESCRITURA DEPENDE DEL CLK POR LO TANTO VA DENTRO DEL PROCESS
   PROCESS (WRITE_ENABLE,CLK,WRITE_DATA,RW_ADRDRESS, RAUX, WAUX, POSINICIAL)
      SUBTYPE REGISTRO IS STD_LOGIC_VECTOR (31 DOWNTO 0);
      TYPE REG_BANK IS ARRAY (0 TO 31) OF REGISTRO;
      VARIABLE ROM_MEMORY: REG_BANK:=  (
                                          X"00000000",
                                          OTHERS => (OTHERS =>'0')
                                       );
   BEGIN
         --write a la memoria depende de la senal de CLK
         IF CLK = '1' AND CLK'EVENT THEN
            IF WAUX = '1' THEN
               ROM_MEMORY((CONV_INTEGER(RW_ADRDRESS) - CONV_INTEGER(POSINICIAL))/4):= WRITE_DATA;
            ELSE NULL;
            END IF;
         ELSE NULL;
         END IF;

         --READ ES AJENO AL PULSO DE RELOJ
         IF RAUX = '1' THEN
            READ_DATA <= ROM_MEMORY((CONV_INTEGER(RW_ADRDRESS) - CONV_INTEGER(POSINICIAL))/4);
         ELSE READ_DATA <= x"00000000"; --cuando ambos (read and write) son 0 o 1 cae en este caso
         END IF;
   END PROCESS;

   --LOS VALORES DE ENABLE NO PUEDEN SER AL MISMO TIEMPO
   RAUX <= (WRITE_ENABLE XOR READ_ENABLE) AND (NOT WRITE_ENABLE);
   WAUX <= (WRITE_ENABLE XOR READ_ENABLE) AND (NOT READ_ENABLE);

end Behavioral;
```

Fig.2.2 Código Fuente de la Memoria

La variable REG_BANK tiene dos parámetros que nos facilitan la creación de una memoria. El primero es la cantidad de palabras que vas a manejar, y el segundo parámetro es el tamaño en bits de las palabras. En este caso, son 32 palabras cada una de 32bits.

De forma similar, los auxiliares nos facilitan que no exista el error de que se active la lectura y la escritura de una palabra. si ambas son iguales XOR nos dará un valor de 0 el cual no permite que se realice ninguna operación. Si son diferentes, se verifica que el valor de un enable sea el opuesto al otro, enseguida ese valor se asigna a la variable de control que va a definir la operación a realizar.

Código Test Bench de Memoria Ram

Leer al menos tres localidades distintas de la memoria de instrucciones. Realizar lectura en al menos dos localidades distintas de la memoria de datos, mostrar el contenido y posteriormente escribir datos nuevos. Al final volver a leer para verificar que el contenido cambió. Recuerda que la lectura no está asociada a un pulso de reloj.

```
60      -- Stimulus process
61      stim_proc: process
62      begin
63         --palabras de 0 a 31
64         -- read la palabra 0 inicial
65         wait for 50ns;
66            WRITE_ENABLE <= '0';
67            READ_ENABLE  <= '1';
68            RW_ADRDRESS <= x"00000000"; --escribir en palabra 4
69            WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
70         -- read la palabra 1 incial
71         wait for 50ns;
72            WRITE_ENABLE <= '0';
73            READ_ENABLE  <= '1';
74            RW_ADRDRESS <= x"00000004"; --escribir en palabra 4
75            WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
76         -- read la palabra 3 inicial
77         wait for 50ns;
78            WRITE_ENABLE <= '0';
79            READ_ENABLE  <= '1';
80            RW_ADRDRESS <= x"0000000C"; --escribir en palabra 4
81            WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
82         -- write a la palabra 1
83         wait for 50ns;
84            WRITE_ENABLE <= '1';
85            READ_ENABLE  <= '0';
86            RW_ADRDRESS <= x"00000004"; --escribir en palabra 4
87            WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
88         --write a la palabra 3
89         wait for 50ns;
90            WRITE_ENABLE <= '1';
91            READ_ENABLE  <= '0';
92            RW_ADRDRESS <= x"0000000C"; --escribir en palabra 4
93            WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
94
```

```
94            _
95      -- read la palabra 0 para verificar que la conversion esta bien
96      wait for 50ns;
97         WRITE_ENABLE <= '0';
98         READ_ENABLE  <= '1';
99         RW_ADRDRESS <= x"00000000"; --escribir en palabra 4
100        WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
101
102     -- read la palabra 1.. el read read de 0 antes para verificar que si se almaceno el valor indefinidamente
103     wait for 50ns;
104        WRITE_ENABLE <= '0';
105        READ_ENABLE  <= '1';
106        RW_ADRDRESS <= x"00000004"; --escribir en palabra 4
107        WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
108
109     -- read la palabra 0 verificar que no paso algo extrano despues de leer el valor que escribimos
110     wait for 50ns;
111        WRITE_ENABLE <= '0';
112        READ_ENABLE  <= '1';
113        RW_ADRDRESS <= x"0000000C"; --escribir en palabra 4
114        WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
115
116     -- Verificar que no se pueda hacer el read y write al mismo tiempo
117     wait for 50ns;
118        WRITE_ENABLE <= '1';
119        READ_ENABLE  <= '1';
120        RW_ADRDRESS <= x"00000000"; --escribir en palabra 4
121        WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
122     wait for 50ns;
123        WRITE_ENABLE <= '0';
124        READ_ENABLE  <= '0';
125        RW_ADRDRESS <= x"00000000"; --escribir en palabra 4
126        WRITE_DATA  <= x"00000001"; -- valor que se escribe en la palabra
127
128     wait;
129  end process;
130
131  END;
```

Resultados de TestBench

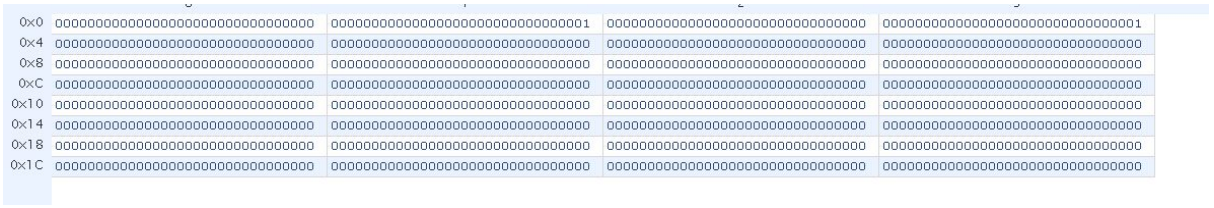

Fig 2.3. Histograma de cambios de las palabras en memoria

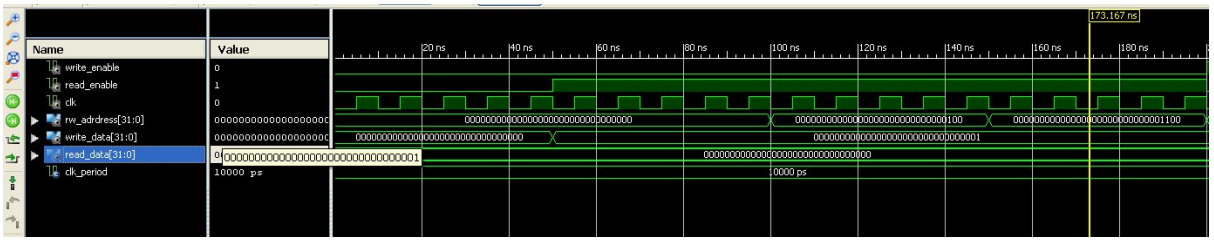

Fig. 2.4 Lectura de valores iniciales de tres palabras

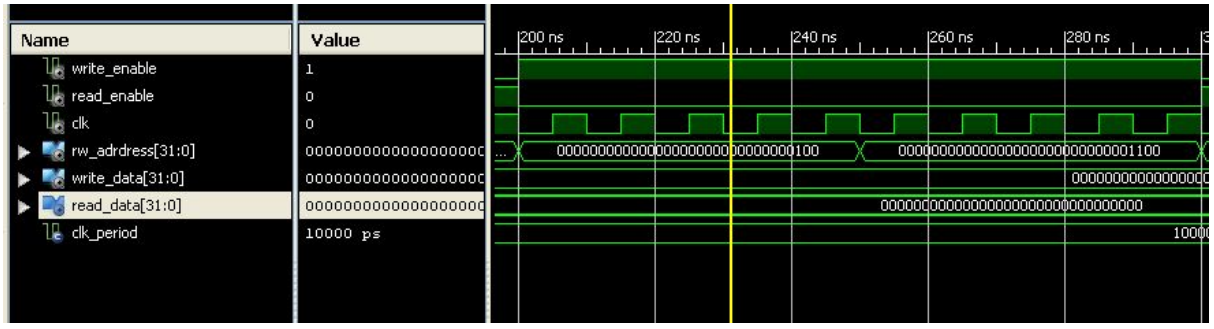

Fig. 2.4 Escritura de valores en las tres palabras

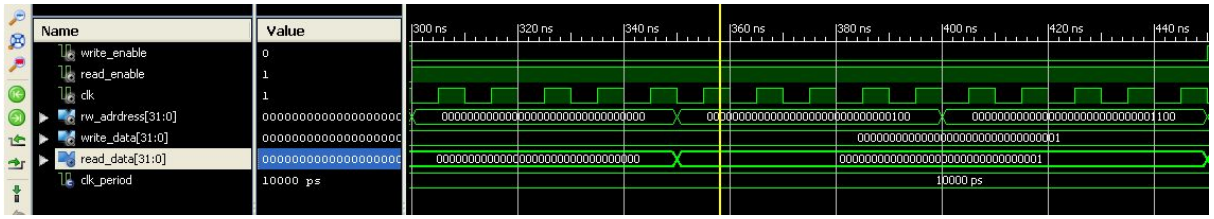

Fig. 2.5 Lectura de palabras con nuevos valores

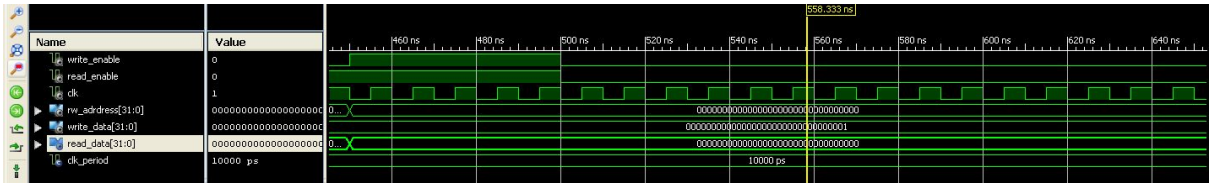

Fig 2.6 Casos especiales ambos 1 ambos 0

Conclusión

Cinthia

En esta práctica en especial nos sentimos un tanto confundidos porque no lográbamos visualizar lo que se solicitaba, además que durante el trayecto de la programación del código, una vez resuelto este inconveniente tuvimos que investigar sintaxis para que nuestro código hiciera lo que deseábamos. Una vez terminado nuestra actividad con los resultados exitosos, me hace sentir finalmente que nuestra primera idea acerca de los conceptos y plantearlo en un código no estaba mal. Aprendí un poco más a detalle sobre el comportamiento de este tipo de memorias, y sé que nos será útil para poder realizar las siguientes prácticas.

Jorge

A lo largo de las prácticas existe un sentimiento que no sabemos lo que estamos realizando, en este caso se presenta un código ejemplo pero para entender la sintaxis de lo que hace el código es necesario invertir tiempo en investigar. De igual manera, en este punto la clase comienza a alcanzar al laboratorio en cuanto al tema, sin embargo, la clase no se explican de forma clara los conceptos sobre todo los que incluyen matemáticas. Nuevamente, mi recomendación es comprar el volumen uno y volumen dos de los libros de Sistemas Digitales Avanzados de la profesora Norma Roffe y en el caso de youtube investigar el funcionamiento del código, ver ejemplos de cómo funciona el modulo que vas a realizar.

Bibliografía

Memoria Ram
https://www.youtube.com/watch?v=LrdEmHoE5U8&ab_channel=twalsh123

Roffe Samaniego, Norma Frida. Diseño de sistemas digitales a través de diseños esquemáticos y VHDL. Volumen 2 (Spanish Edition) . American Academy of Pediatrics. Kindle Edition.

Roffe Samaniego, Norma Frida. Diseño de sistemas digitales a través de diseños esquemáticos y VHDL. Volumen 1 (Spanish Edition) . American Academy of Pediatrics. Kindle Edition.