Benemérita Universidad Autónoma de Puebla

Facultad de ciencias de la computación

Arquitectura de computadoras – Práctica 5

Alumnos:

Amador Ortega Christian Amauri - 201927821

Bryan Arturo Cedillo García – 201934964

Docente: Lilia Mantilla Narváez

Introducción:

En el campo de la arquitectura de computadores, la optimización del rendimiento y la eficiencia de los sistemas es un objetivo fundamental. Una técnica ampliamente utilizada para mejorar el rendimiento en la ejecución de instrucciones es la carga segmentada. Esta estrategia permite aprovechar al máximo los recursos de los procesadores, alentando el flujo constante de instrucciones y datos a través de la canalización, lo que resulta en un aumento significativo en el rendimiento general.

Es importante destacar que se requiere una memoria de datos eficiente y optimizada para respaldar esta técnica. La memoria de datos desempeña un papel crucial al almacenar y proporcionar acceso a los datos necesarios para la ejecución de instrucciones en diferentes etapas de la carga segmentada.

Marco teórico:

La operación de carga segmentada en el campo de la arquitectura de computadores se basa en varios conceptos fundamentales. A continuación, se presentan 3 aspectos clave:

- Dependencia de datos: En la ejecución de instrucciones, las dependencias de datos se refieren a las relaciones entre instrucciones donde una instrucción depende de los resultados de otra instrucción previa. Estas dependencias deben ser gestionadas adecuadamente para evitar conflictos y garantizar el orden correcto de ejecución de las instrucciones.

- Memoria de datos: La memoria de datos es esencial para la operación de carga segmentada, ya que almacena los datos requeridos por las instrucciones durante su ejecución. Una memoria de datos eficiente y bien diseñada permite un acceso rápido y oportuno a los datos, minimizando los tiempos de espera y optimizando el rendimiento general del sistema.

- Unidad de control: En la operación de carga segmentada, una unidad de control es necesaria para coordinar y controlar el flujo de instrucciones a través de las diferentes etapas de la canalización. La unidad de control se encarga de secuenciar y sincronizar las instrucciones, detectar y resolver conflictos y gestionar las dependencias entre instrucciones.

Desarrollo:

El camino de datos se reutilizó de la práctica 4 (con un par de componentes ligeramente actualizados pero equivalentes) Y se añadieron 4 tipos de componentes (un generador de pulsos, una memoria de datos, un contador de 5 bits y un multiplexor). (Se tenía la intención de crear un quinto componente, una pequeña unidad de control, que *iba a manejar la segmentación del camino de datos* pero por razones de tiempo, no pudo ser llevada a cabo)

A continuación presentaremos los códigos de los 4 componentes descritos, y la configuración que se llevó a cabo para ellos.

Generador de pulsos

GeneradorDePulsos.vhdl:

Simplemente tiene una entrada de reloj diseñada para que al introducir un reloj de 1 a 0, mande 4 salidas diferentes (1,0,1 a 0 y 0 a 1) para cubrir y automatizar diferentes necesidades de diferentes componentes al mismo tiempo y con un mismo orden de inicio. (la única señal sin usar, fue '1').

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity GeneradorDePulsos is
port (
    clk : in std_logic;
    pulso_1 : out std_logic;
    pulso_0 : out std_logic;
    pulso_1_a_0 : inout std_logic;
    pulso_0_a_1 : inout std_logic
  );
end GeneradorDePulsos;

architecture Behavioral of GeneradorDePulsos is
signal a : std_logic;
signal b : std_logic;
begin

      process(clk)
      begin
            if clk'event then
                  pulso_1 <= '1';
                  pulso_0 <= '0';
                  pulso_1_a_0 <= '1';
                  pulso_0_a_1 <= '0';
                  a <= pulso_1_a_0;
                  b <= pulso_0_a_1;
                  if rising_edge(clk) then
```

```
                        pulso_1_a_0 <= not a;
                        pulso_0_a_1 <= not b;
                    end if;
                end if;
        end process;
end Behavioral;
```

Memoria de datos

MemoriaDeDatosV2.vhdl

Implementamos 2 versiones de una memoria de datos RAM (Random Access Memory) de 32 vectores de 32 bit para guardar los datos trabajados, nos quedamos con la segunda porque era más compacta y versátil que la primera. Esta memoria lee el dato recibido en la entrada DW cuando la señal “MemWrite” es igual a ‘0’ y lo guarda en la posición indicada por la entrada “ADDR” de la memoria RAM. Y manda lo que hay en la posición indicada por la entrada “ADDR” cuando la señal “MemRead” es igual a ‘1’. (estas definiciones de señales son ajustables y flexibles, de hecho se modificaron muchas veces para explorar varias combinaciones)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MemoriaDeDatosv2 is
    port (
        Addr : in STD_LOGIC_VECTOR (4 downto 0); -- Direccin de memoria (0 a 31)
        DW : in STD_LOGIC_VECTOR (31 downto 0); -- Datos a escribir
        MemWrite : in STD_LOGIC; -- Señal de escritura
        MemRead : in STD_LOGIC; -- Señal de lectura
        DR : out STD_LOGIC_VECTOR (31 downto 0) -- Datos leídos
    );

end MemoriaDeDatosv2;

architecture Behavioral of MemoriaDeDatosv2 is
    type ram_type is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
    signal RAM : ram_type := (others => (others => '0'));
begin

    process (MemRead, MemWrite, Addr, DW)
    begin
        if MemRead = '1' then --salida de datos
            DR <= RAM(conv_integer((Addr)));
        elsif MemWrite = '0' then --entrada de datos
            RAM(conv_integer((Addr))) <= DW;
        end if;
    end process;
end Behavioral;
```

Contador de 5 bits

Contador_5bits.vhdl

Simplemente es un contador auxiliar (a falta de una unidad de control que regule esta y otras operaciones), que fue diseñado específicamente para trabajar con la señal de entrada ADDR de la memoria de datos (para potenciales operaciones de lectura y escritura). (incrementa un valor de 5 bits en 1, cada vez que la señal de entrada es activada (en este caso, la señal de entrada será la operación de lectura de la memoria de datos)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Contador_5bits is
    Port (
        clk : in std_logic;
        salida : out std_logic_vector(4 downto 0)
    );

end Contador_5bits;


architecture Behavioral of Contador_5bits is
    signal contador : std_logic_vector(4 downto 0) := (others => '0');
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if contador = "11111" then
                contador <= (others => '0');
            else
                contador <= std_logic_vector(contador + 1);
            end if;
        end if;
    end process;
    salida <= contador;
end Behavioral;
```

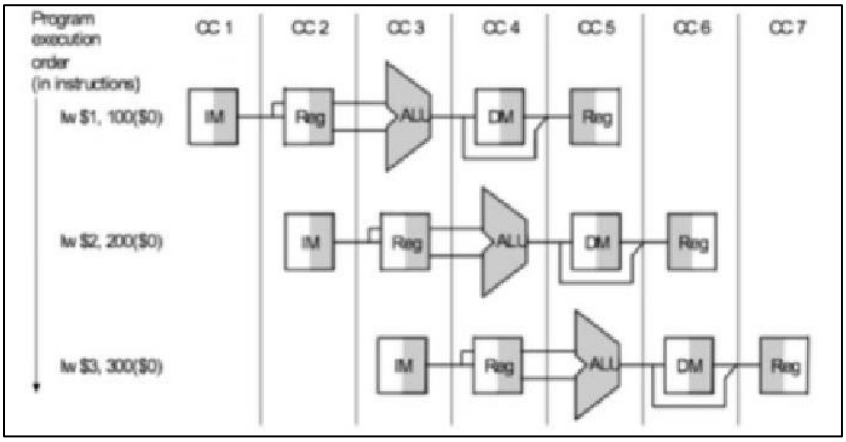
Multiplexor

MUX_MEMREGISTROS.vhdl

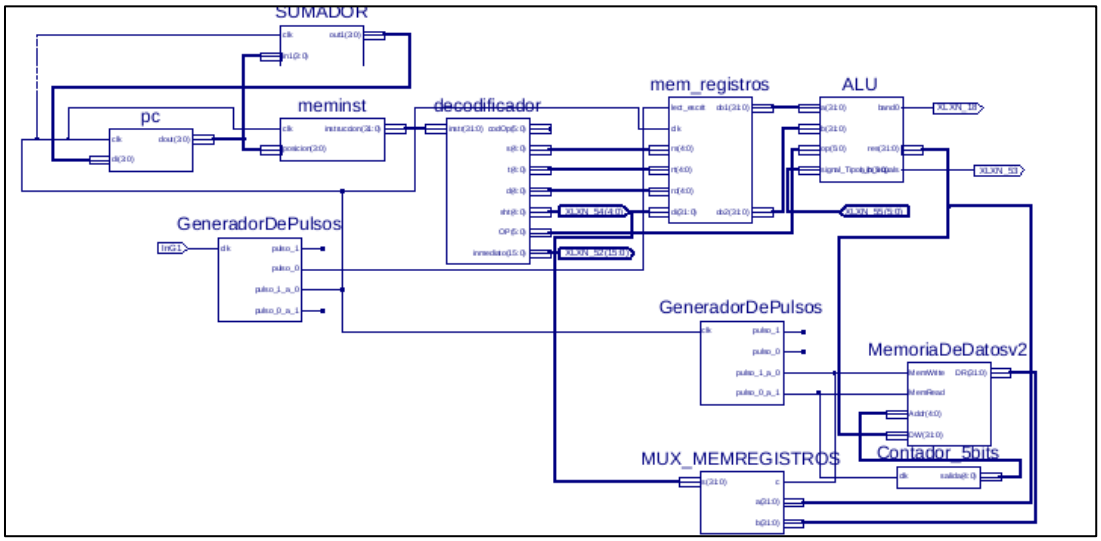
Es un multiplexor básico, dos entradas de 32 bits que representan los datos a escoger (a y b), una entrada de un bit para escoger una de las otras dos entradas (c) y una salida de 32 bits (s). La entrada “a” corresponde a la salida directa del resultado de la ALU. La entrada “b” corresponde a la salida de la memoria de datos. (salida obtenida de anteriores operaciones y de una señal de una posición de memoria a obtener).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_MEMREGISTROS is
	port(
		a : in std_logic_vector (31 downto 0);
		b : in std_logic_vector (31 downto 0);
		c : in std_logic;
		s : out std_logic_vector (31 downto 0)
	);
end MUX_MEMREGISTROS;

architecture Behavioral of MUX_MEMREGISTROS is
begin

process(a,b,c)
begin
	if (c = '0') then
		s <= a;
	else
		s <= b;
	end if;
end process;
end Behavioral;
```

Dado el diagrama:



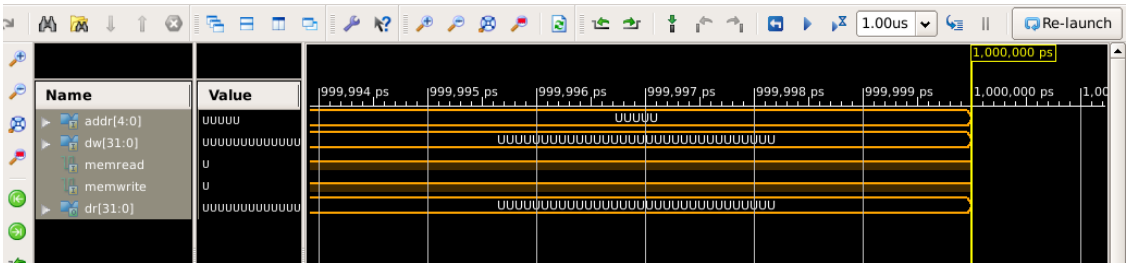
El diagrama esquemático que implementamos sigue el camino: memoria de instrucciones (y decodificador), memoria de registros, ALU, memoria de datos y memoria de registros (en esta ultima parte creamos un bucle donde la memoria de registros es retroalimentada por la ALU o la memoria de datos, dependiendo del estado de la bandera ‘c’ del multiplexor “MUX_MEMREGISTROS”):



(el generador de pulsos principal alimenta al generador de pulsos del bloque de carga de datos para que se ejecuten en base a una misma secuencia)

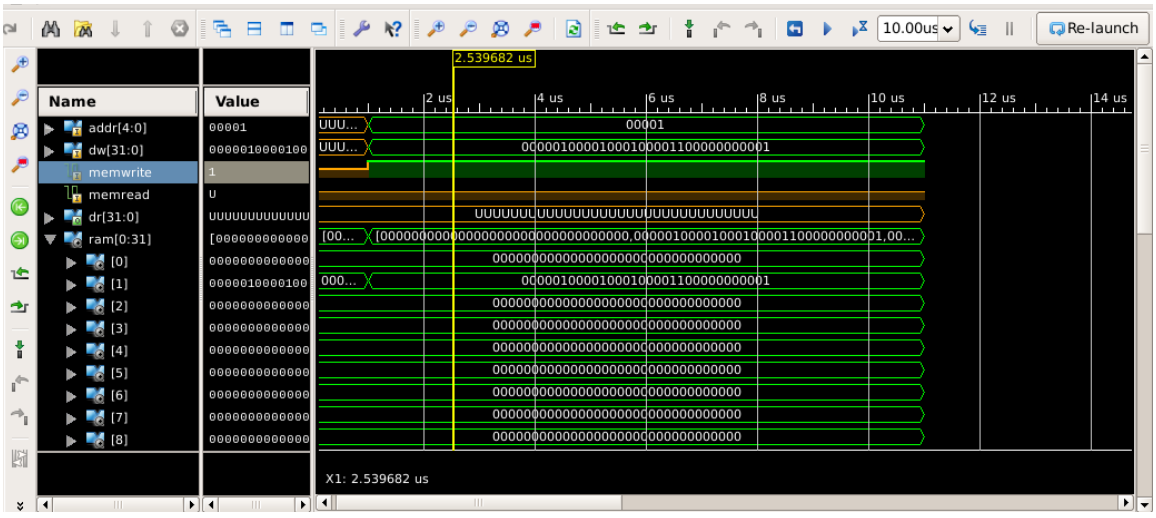
Resultado:

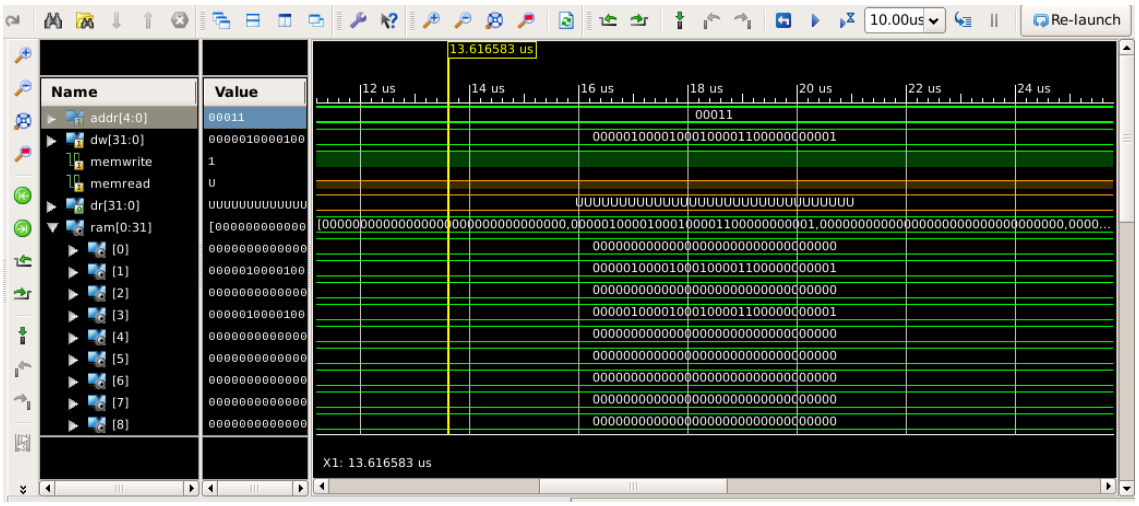
Al hacer la simulación del esquemático de la memoria de datos, no se puede ver directamente los datos que se han almacenado (el esquemático no los muestra) (no hay un array, de arrays):

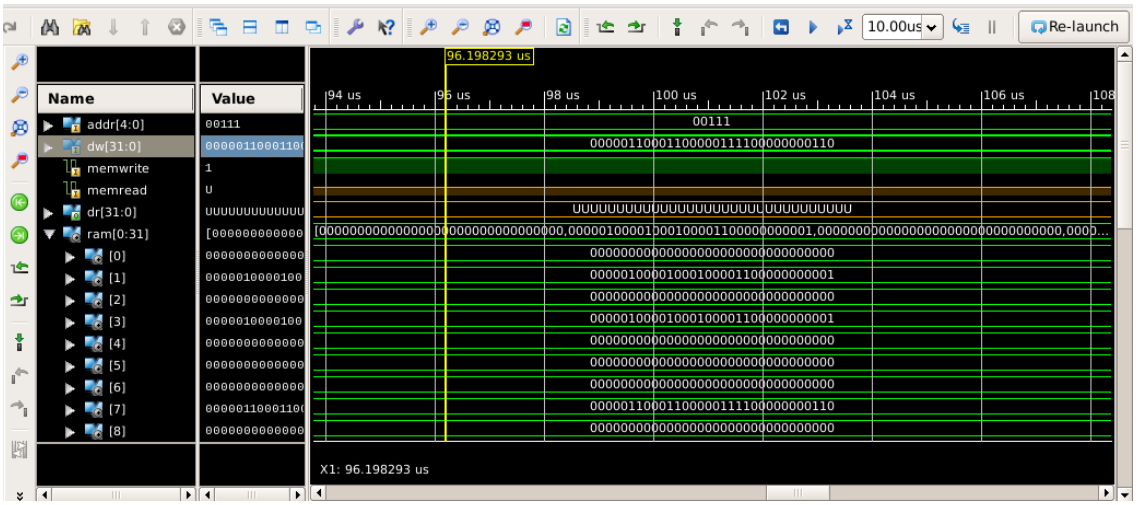


Pero podemos confiar en el diseño que implementamos para ella porque la simulación del componente (el .vhdl)sí los muestra (ram es el array de arrays):

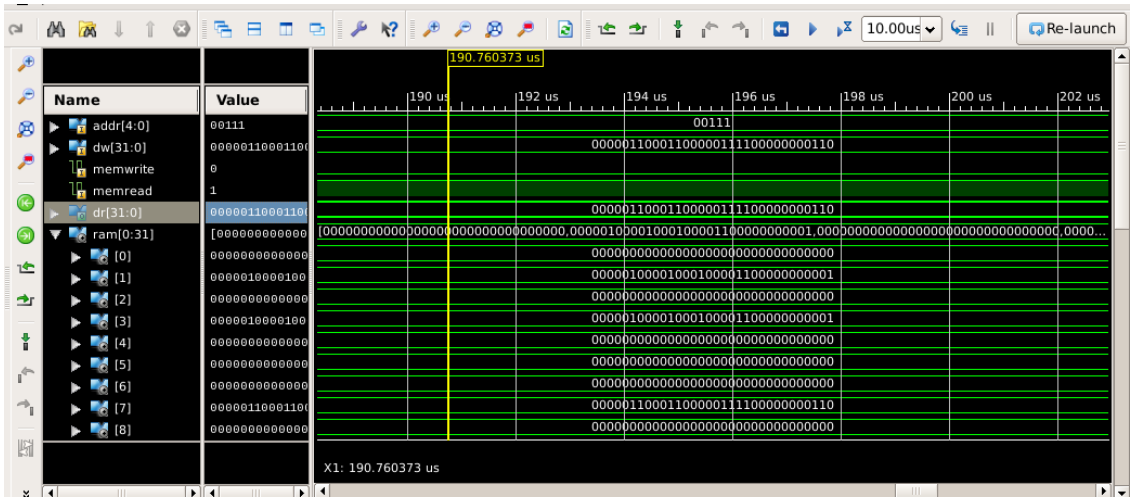
Dadas las entradas en ADDR y memwrite, el dato ingresado se guarda en la posición indicada en la ram (como mencionamos, las señales memread y memwrite pueden ser modificadas a convenciencia antes de la ejecución. en esta simulación, la señal de activación de memwrite es 1):







Y si la señal memread es ‘1’, el dato indicado por la posición “ADDR” sale por el puerto “DR”



Simulación del camino completo:

Hay muchas señales en la simulación, pero las de interés son las que tienen un nombre personalizado (no por default) Es decir, como se ve en la imagen de ing1, a meminstout. Siendo estos sus significados:

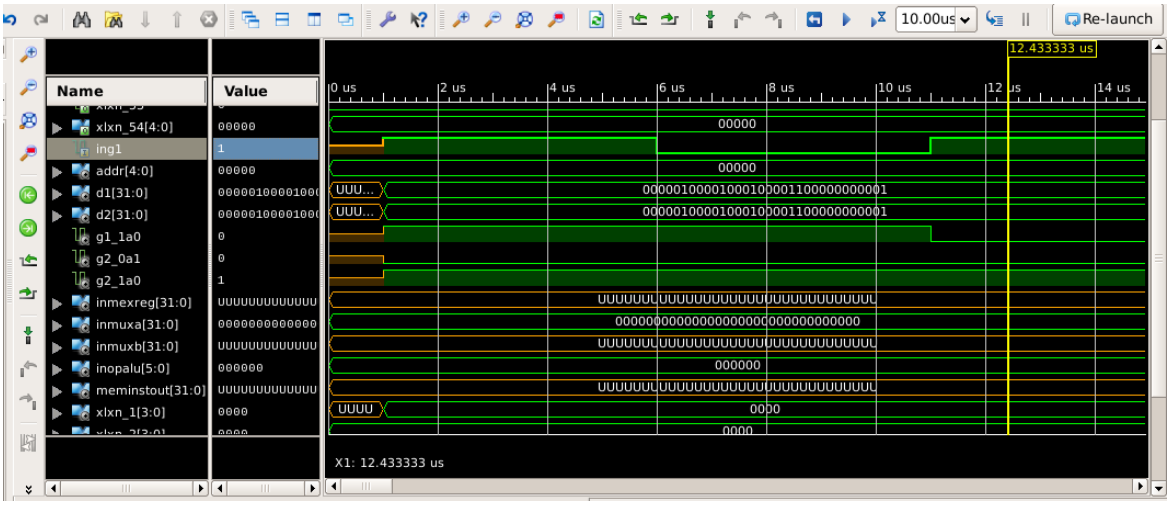
- Ing1: Entrada del generador de señales principal (este debe ser 1 a 0 para automatizar todas las señales) (clk)
- ADDR: Salida del contador de 5 bits. indica la dirección de acceso de la memoria de datos. (5 bits)
- D1 y D2: salidas de la memoria de registros. son los dos datos a operar. (32 bits)
- g1_1a0, g2_1a0 y g2_0a1: son las salidas de los generadores de señales (generador 1 de 1 a 0 ,generador 2 de 1 a 0 y generaor 2 de 0 a 1 respectivamente).
- g1_1a0 usado para: entrada clk del segundo generador de pulsos, pc, meminst, sumador y mem_registros.
- g2_1a0 usado para: entrada MemWrite de la memoria de datos y entrada c de MUX_REGISTROS.
- g2_0a1 usado para: entrada clk del contador de 5 bits y entrada MemRead de la memoria de datos.
- Inmexreg: es la entrada a di de la memoria de registros (32 bits)
- Inmuxa: es la entrada “a” del multiplexor, cuyo valor se obtiene de la salida del resultado de la ALU (dicho resultado también se usa para la entrada DW de la memoria de datos) (32 bits)

- Inmuxb: es la entrada “b” del multiplexor, cuyo valor se obtiene de la salida de la consulta a la memoria de instrucciones. (32 bits)

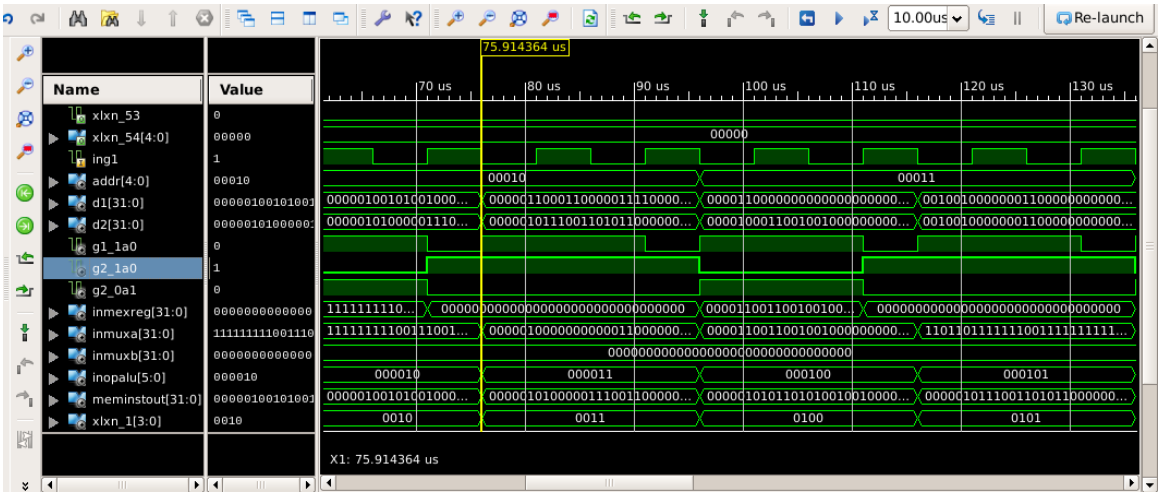
- Inopalu: salida del decodificador de instrucciones, representa el código de instrucción a ingresar en la ALU. (6 bits)

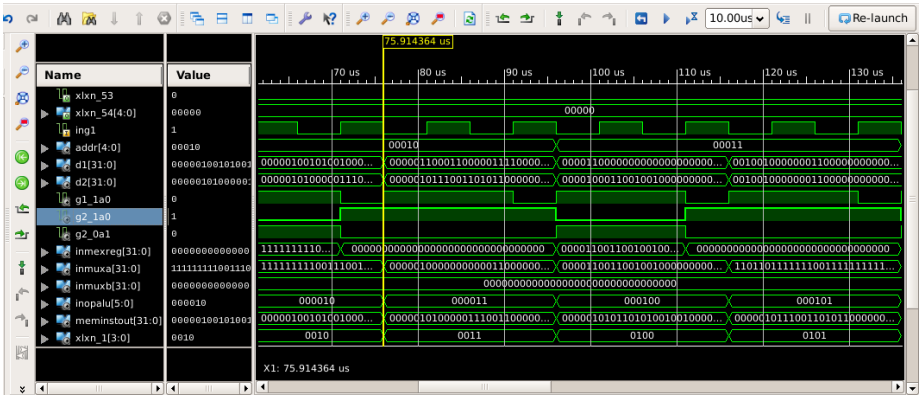
- Meminstout: salida de la memoria de instrucciones (32 bits, manda una instrucción)

Al no tener una unidad de control las señales al inicio toman diferentes ritmos:



Unos cuantos saltos después, las cargas de datos a la memoria de registros tardan 20 us en completarse





Si la señal g2_1a0 es ‘0’, (MemWrite = 0) el dato que sale por InMexReg, tomará el valor de la entrada “a” del multiplexor, que corresponde a la salida directa de la ALU. Y al mismo tiempo se debe guardar ese dato en la memoria de datos

Luego, si g2_0a1 es ‘1’ el dato de salida deberá tomar el valor de la entrada “b”, es decir, toma el valor desde la memoria de datos. (la posición ADDR de la memoria de datos).

(Y claro, si un estado está activado, el otro está desactivado).

En general la entrada y salida de los datos es correcta sólo en el sentido de que los componentes del sistema se están comunicando “exitosamente”. Pero no los datos no son los deseados porque no tienen orden, requieren una secuencia. Secuencia que puede ser implementada mediante ***segmentación*** *con una unidad de control que regule los estados adecuados para cada parte del camino. En un futuro (esperemos no demasiado lejano), esto será así….*

Conclusiones:

Esta práctica ha resaltado la importancia de la unidad de control en el flujo de instrucciones. La unidad de control coordina y sincroniza las etapas de la canalización, detectando conflictos y gestionando dependencias. Sin una unidad de control adecuada, se presentan limitaciones en la ejecución, como conflictos y rendimiento subóptimo. Es crucial reconocer el papel clave de la unidad de control para asegurar una ejecución eficiente y minimizar limitaciones en la operación de carga segmentada.

Además, una buena implementación de la memoria de datos es fundamental para garantizar un acceso rápido y eficiente a los datos requeridos por las instrucciones en cada etapa de la carga segmentada. Un diseño óptimo de la memoria de datos puede mejorar significativamente el rendimiento del sistema al reducir los tiempos de espera y minimizar los cuellos de botella durante la ejecución de instrucciones.

Bibliografía:

- Harris, D., & Harris, S. L. (2013). Digital design and computer architecture. Morgan Kaufmann Publishers.

- Patterson, D. A., & Hennessy, J. L. (2018). Computer organization and design RISC-V edition: The hardware software interface. Morgan Kaufmann Publishers.