Universidad Nacional Autónoma de México

Facultad de Ciencias

Organización y Arquitectura de Computadoras
2025-2

Práctica 04

Docentes:
José Galaviz Ricardo Pérez Ximena Lezama

Autores:
Fernanda Ramírez Juárez Ianluck Rojo Peña

Fecha de entrega: Jueves 06 de marzo de 2025

Preguntas.

1. ¿Qué es VHDL? ¿Cómo se ve un full-adder en VHDL? y ¿Cómo se ve un Half-Adder?

   Aviso. El contenido de la respuesta a esta pregunta fue obtenida de la siguiente página:
   How to Implement a Full Adder in VHDL
   VHDL *(La V significa **VHSIC (Very High Speed Integrated Circuits)**, VHSIC Hardware Description Language)* es un lenguaje de descripción de circuitos electrónicos digitales que utiliza distintos niveles de abstracción. Permite acelerar el proceso de diseño. No es un lenguaje de programación, por ello conocer su sintaxis no implica necesariamente saber diseñar con él, en cambio VHDL es un lenguaje de descripción de hardware, que permite describir circuitos síncronos y asíncronos. En particular permite tanto una descripción de la estructura del circuito, como la especificación de la funcionalidad de un circuito utilizando formas familiares a los lenguajes de programación.

   A continuación se muestra un ejemplo del sumador completo. En el código VHDL, este está implementado en la línea 24 sobre la entrada registrada. Nótese que, antes de realizar la operación de suma, se extiende el número de bits del operando de entrada. Esto se implementa utilizando la función estándar "resize" proporcionada en el paquete $\backslash numeric_std$", como se muestra en las líneas 31 y 32.

```
1       library ieee;
2       use ieee.std_logic_1164.all;
3       use ieee.numeric_std.all;
4
5       entity adder_full_signed_reg is
6       generic(
7         N             : integer := 8);
8       port (
9         i_clk        : in     std_logic;
10        i_add1       : in     std_logic_vector(N-1 downto 0);
```

```
    i_add2      : in     std_logic_vector(N-1 downto 0);
    o_sum       : out    std_logic_vector(N downto 0));
end adder_full_signed_reg;

architecture rtl of adder_full_signed_reg is

signal r_add1   : signed(N downto 0);
signal r_add2   : signed(N downto 0);
signal w_sum    : signed(N downto 0);

begin

-- combinatorial adder
  w_sum <= r_add1 + r_add2;

r_process : process(i_clk)
begin
  if(rising_edge(i_clk)) then

  -- register input and extend sign
    r_add1      <=  resize(signed(i_add1),N+1);
    r_add2      <=  resize(signed(i_add2),N+1);

  -- register output
    o_sum       <= std_logic_vector(w_sum);

  end if;
end process r_process;

end rtl;
```

Listing 1: Código VHDL - Sumador con Registro

En el ejemplo, se implementa un sumador completo con signo. Para un sumador completo sin signo, solo es cuestión de reemplazar la línea que contiene la definición del tipo de datos “signed” por “unsigned”.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder_full_unsigned is
generic(
  N         : integer := 8);
port (
  add1      : in     std_logic_vector(N-1 downto 0);
  add2      : in     std_logic_vector(N-1 downto 0);
  sum       : out    std_logic_vector(N downto 0));
end adder_full_unsigned;

architecture rtl of adder_full_unsigned is

begin

  sum <= std_logic_vector( resize(unsigned(add1),N+1) + resize(
```

```
            unsigned(add2),N+1) );

    end rtl;
```

Listing 2: Código VHDL - Sumador sin Registro

En este caso, el sumador es puramente combinacional. VHDL código para Half Adder:

```
    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

    entity adder_half_unsigned_reg is
    generic(
      N       : integer := 256);
    port (
      i_clk     : in     std_logic;
      i_add1    : in     std_logic_vector(N-1 downto 0);
      i_add2    : in     std_logic_vector(N-1 downto 0);
      o_sum     : out    std_logic_vector(N-1 downto 0));
    end adder_half_unsigned_reg;

    architecture rtl of adder_half_unsigned_reg is

    signal r_add1    : unsigned(N-1 downto 0);
    signal r_add2    : unsigned(N-1 downto 0);
    signal w_sum     : unsigned(N-1 downto 0);

    begin

    -- combinatorial adder
      w_sum <= r_add1 + r_add2;

    r_process : process(i_clk)
    begin
      if(rising_edge(i_clk)) then

      -- register input
        r_add1      <=  unsigned(i_add1);
        r_add2      <=  unsigned(i_add2);

      -- register output
        o_sum       <= std_logic_vector(w_sum);

      end if;
    end process r_process;

    end rtl;
```

Listing 3: Código VHDL - Medio Sumador

2. En el contexto de las ALUs, explica qué son los: SIMD, MIMD, La Execution Units (EUs) y los Address Generation Units (AGUs).

En el contexto de las ALUs (Unidades Aritmético-Lógicas), se utilizan diversas técnicas y unidades especializadas para aumentar el rendimiento y el paralelismo.

- **SIMD (Single Instruction, Multiple Data):** Técnica de procesamiento paralelo en la que una sola instrucción se aplica simultáneamente a múltiples datos. Permite que una ALU procese varios elementos de datos en paralelo, aumentando la eficiencia en tareas que involucran operaciones repetitivas. Muy útil en el procesamiento de vectores, imágenes, audio o cualquier tarea donde la misma operación se deba realizar sobre muchos elementos. Es clave en unidades como AVX en CPUs y en GPUs modernas.
- **MIMD (Multiple Instruction, Multiple Data):** Arquitectura en la que múltiples procesadores ejecutan diferentes instrucciones sobre distintos datos de manera independiente. Facilita la multitarea y la ejecución concurrente de programas complejos, mejorando el rendimiento en entornos con cargas de trabajo heterogéneas. Es fundamental para la computación paralela, con aplicaciones en simulaciones, modelado y diseño asistido.
- **Execution Units (EUs):** Unidades dentro del procesador responsables de ejecutar instrucciones. Pueden incluir ALUs y otras unidades especializadas en operaciones de punto flotante o enteras. Durante la ejecución de un programa, las EUs reciben las instrucciones decodificadas y realizan los cálculos o las operaciones correspondientes. Permiten la ejecución paralela y especializada de distintas operaciones, lo que optimiza el rendimiento general del procesador mediante técnicas como la ejecución en paralelo y el pipelining.
- **Address Generation Units (AGUs):** Unidades dedicadas al cálculo de direcciones de memoria necesarias para las operaciones de carga y almacenamiento, optimizando el acceso a los datos. Cuando se necesita acceder a una posición de memoria, las AGUs calculan la dirección efectiva a partir de una base, un offset y, en ocasiones, una escala. Al separar el cálculo de direcciones de las operaciones aritméticas y lógicas, se agiliza el acceso a la memoria, permitiendo que la CPU opere de manera más eficiente al realizar múltiples tareas en paralelo.

  En conjunto, estos elementos permiten mejorar la eficiencia del procesamiento de datos y la ejecución de instrucciones en arquitecturas modernas de computación.

3. ¿Qué es el Carry-Look-Ahead Adder?

El *Carry-Look-Ahead Adder* ***(Sumador de Acarreo Anticipado, CLA)*** es un tipo de sumador digital diseñado para realizar operaciones aritméticas de forma rápida y eficiente, reduciendo el tiempo de espera asociado a la propagación del acarreo en sumadores tradicionales como el *Ripple Carry Adder*.

Características:

- Se introdujo en 1958 y ha sido la base de varias modificaciones e implementaciones en circuitos digitales.
- Calcula uno o más bits de acarreo antes de la suma, evitando la propagación secuencial del acarreo.

- Reduce el tiempo de propagación, lo que permite una ejecución más rápida de operaciones aritméticas.
- Se utiliza en diseños de procesadores, especialmente en operaciones de multiplicación y división.

Funcionamiento: El CLA emplea dos señales clave para determinar cómo se propagan los acarreos:

- Carry Generate (G): Indica que una posición generará un acarreo independientemente del acarreo de entrada.
- Carry Propagate (P): Indica que un acarreo de entrada se propagará a la siguiente posición.

Para cada posición de bit, el CLA analiza si se generará un acarreo o si se propagará el acarreo desde una posición anterior. Al calcular estos valores en paralelo, el CLA puede deducir rápidamente los acarreos sin esperar a que se propaguen bit por bit.

Ventajas:

- Mayor velocidad. Reduce el retraso de propagación del acarreo, acelerando las sumas de múltiples bits.
- Mejor rendimiento en circuitos de gran tamaño. Especialmente útil en ALUs.
- Optimización con unidades de acarreo anticipado (LCU). Se utilizan junto con el CLA para calcular acarreos de manera más eficiente en sumadores de mayor tamaño.

Punto Extra.

Link al video para el punto extra: *Compuertas Lógicas en Minecraft*