1) Elementos de los LEs y estructura de las LABs en FPGA Cyclone III

Los Logic Elements (LEs) constituyen la unidad lógica básica de la arquitectura Cyclone III. Cada LE está formado por una LUT de 4 entradas capaz de implementar cualquier función lógica de hasta cuatro variables, junto con un flip-flop programable que permite el almacenamiento de datos. Además, cada LE incorpora multiplexores especializados para las cadenas de carry y cascade, que facilitan la implementación eficiente de operaciones aritméticas y funciones combinacionales complejas respectivamente. La lógica de control asociada gestiona las señales de clock, clear y enable del registro.

En cuanto a las LABs (Logic Array Blocks), estas agrupan 16 Logic Elements formando bloques funcionales más grandes. Cada LAB comparte señales de control comunes entre sus LEs, lo que optimiza el uso de recursos para implementar circuitos con clock, clear y enable compartidos. Las interconexiones locales dentro del LAB permiten comunicación rápida entre los LEs adyacentes, mientras que las conexiones a la red de interconexión global posibilitan la comunicación con otros LABs del dispositivo. Las cadenas de carry y cascade atraviesan completamente el LAB, permitiendo construir sumadores y funciones lógicas anchas de manera eficiente.

2) ¿De qué se trata el Nios® II?

El Nios® II es un procesador soft-core de 32 bits desarrollado por Altera (actualmente parte de Intel) específicamente diseñado para ser implementado en FPGAs de la familia Altera. A diferencia de los procesadores tradicionales, este procesador se sintetiza utilizando los recursos lógicos programables del dispositivo, lo que permite una gran flexibilidad en su configuración y uso. La arquitectura RISC de 32 bits del Nios® II está disponible en tres variantes que ofrecen distintos compromisos entre área y rendimiento: la versión Economy optimizada para bajo consumo de recursos, la versión Standard que balancea tamaño y velocidad, y la versión Fast diseñada para aplicaciones que requieren máximo rendimiento.

Este procesador permite desarrollar sistemas embebidos completos dentro de la FPGA, conocidos como SoPC (System on Programmable Chip), donde pueden integrarse periféricos estándar o diseñados a medida según las necesidades específicas de la aplicación. El Nios® II incluye un conjunto completo de herramientas de desarrollo, incluyendo compiladores de C/C++ y soporte para sistemas operativos en tiempo real, lo que facilita el desarrollo de aplicaciones complejas sin necesidad de hardware externo adicional.

3) Diferencia entre IP cores y bloques embebidos

La distinción fundamental entre los IP cores y los bloques embebidos radica en su naturaleza física y en cómo se implementan dentro de la FPGA. Los IP cores son módulos de diseño reutilizables que se sintetizan utilizando los recursos lógicos programables del dispositivo, específicamente los Logic Elements, las interconexiones y la memoria distribuida. Al ser

implementados en lógica reconfigurable, estos cores consumen recursos generales de la FPGA y ofrecen gran flexibilidad, ya que pueden ser modificados, parametrizados o reemplazados según las necesidades del diseño. Ejemplos típicos incluyen controladores de comunicación, procesadores como el Nios II, o bloques de procesamiento de señales implementados en lógica programable.

Por otro lado, los bloques embebidos son circuitos físicos especializados que fueron fabricados directamente en el silicio durante el proceso de manufactura del chip. Estos bloques de hardware dedicado no consumen recursos de Logic Elements y no son reconfigurables en su arquitectura interna, aunque sí pueden configurarse en sus modos de operación. Los bloques embebidos ofrecen ventajas significativas en términos de rendimiento, consumo energético y área ocupada para las funciones específicas que implementan. En la familia Cyclone III encontramos ejemplos como los multiplicadores embebidos de 18x18 bits, los bloques de memoria M9K, y los PLLs para generación de relojes. La decisión de usar IP cores o bloques embebidos depende de la disponibilidad de estos últimos y de los requerimientos específicos de la aplicación en términos de rendimiento, flexibilidad y consumo de recursos.

4) Tipo de celda de programación en FPGA Cyclone III

La FPGA Cyclone III utiliza celdas de programación basadas en tecnología SRAM (Static Random Access Memory). Esta tecnología de configuración presenta características particulares que definen el comportamiento del dispositivo. Al tratarse de memoria volátil, la configuración se pierde cuando el dispositivo es desconectado de la alimentación, por lo que es necesario recargar la configuración desde una memoria externa no volátil (típicamente memoria Flash) cada vez que el sistema se enciende. Este proceso de configuración, aunque necesario en cada encendido, es relativamente rápido comparado con otras tecnologías.

Una ventaja importante de las celdas basadas en SRAM es que permiten la reprogramación ilimitada del dispositivo sin degradación, lo que resulta especialmente útil durante las etapas de desarrollo y depuración de diseños. Durante la operación normal, después de la configuración inicial, el consumo de energía asociado a estas celdas es bajo. La arquitectura de configuración basada en SRAM también facilita la implementación de técnicas de reconfiguración parcial, aunque esta característica está más desarrollada en familias superiores de FPGAs.

5) Descripción VHDL de un Flip Flop JK

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

entity FlipFlop_JK is
    Port (
        J     : in  STD_LOGIC;
        K     : in  STD_LOGIC;
        CLK   : in  STD_LOGIC;
        RESET : in  STD_LOGIC;
```

```
        Q     : out STD_LOGIC;
        Q_n   : out STD_LOGIC
    );
end FlipFlop_JK;

architecture Behavioral of FlipFlop_JK is
    signal Q_temp : STD_LOGIC := '0';
begin
    process(CLK, RESET)
    begin
        if RESET = '1' then
            Q_temp <= '0';
        elsif rising_edge(CLK) then
            if J = '0' and K = '0' then
                Q_temp <= Q_temp;  -- Mantiene el estado
            elsif J = '0' and K = '1' then
                Q_temp <= '0';     -- Reset
            elsif J = '1' and K = '0' then
                Q_temp <= '1';     -- Set
            else  -- J = '1' and K = '1'
                Q_temp <= not Q_temp;  -- Toggle
            end if;
        end if;
    end process;

    Q <= Q_temp;
    Q_n <= not Q_temp;

end Behavioral;
```

6) Descripción VHDL de un sumador completo de un bit

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

entity sumador_completo is
    Port (
        A    : in  STD_LOGIC;
        B    : in  STD_LOGIC;
        Cin  : in  STD_LOGIC;
        Sum  : out STD_LOGIC;
        Cout : out STD_LOGIC
    );
end sumador_completo;

architecture Behavioral of sumador_completo is
begin
    Sum  <= A xor B xor Cin;
    Cout <= (A and B) or (A and Cin) or (B and Cin);
end Behavioral;
```

7) Descripción VHDL del test bench del sumador completo

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity tb_sumador_completo is
end tb_sumador_completo;

architecture Behavioral of tb_sumador_completo is
    -- Declaración del componente a probar
    component sumador_completo
        Port (
            A    : in  STD_LOGIC;
            B    : in  STD_LOGIC;
            Cin  : in  STD_LOGIC;
            Sum  : out STD_LOGIC;
            Cout : out STD_LOGIC
        );
    end component;

    -- Señales de prueba
    signal A_tb    : STD_LOGIC := '0';
    signal B_tb    : STD_LOGIC := '0';
    signal Cin_tb  : STD_LOGIC := '0';
    signal Sum_tb  : STD_LOGIC;
    signal Cout_tb : STD_LOGIC;

begin
    -- Instanciación del sumador completo
    UUT: sumador_completo
        port map (
            A    => A_tb,
            B    => B_tb,
            Cin  => Cin_tb,
            Sum  => Sum_tb,
            Cout => Cout_tb
        );

    -- Proceso de estímulo
    stimulus: process
    begin
        -- Probar todas las combinaciones posibles (8 casos)

        -- Caso 0: 0 + 0 + 0 = 0
        A_tb <= '0'; B_tb <= '0'; Cin_tb <= '0';
        wait for 10 ns;

        -- Caso 1: 0 + 0 + 1 = 1
        A_tb <= '0'; B_tb <= '0'; Cin_tb <= '1';
        wait for 10 ns;

        -- Caso 2: 0 + 1 + 0 = 1
        A_tb <= '0'; B_tb <= '1'; Cin_tb <= '0';
        wait for 10 ns;

        -- Caso 3: 0 + 1 + 1 = 10 (Sum=0, Cout=1)
        A_tb <= '0'; B_tb <= '1'; Cin_tb <= '1';
        wait for 10 ns;

        -- Caso 4: 1 + 0 + 0 = 1
```

```
        A_tb <= '1'; B_tb <= '0'; Cin_tb <= '0';
        wait for 10 ns;

        -- Caso 5: 1 + 0 + 1 = 10 (Sum=0, Cout=1)
        A_tb <= '1'; B_tb <= '0'; Cin_tb <= '1';
        wait for 10 ns;

        -- Caso 6: 1 + 1 + 0 = 10 (Sum=0, Cout=1)
        A_tb <= '1'; B_tb <= '1'; Cin_tb <= '0';
        wait for 10 ns;

        -- Caso 7: 1 + 1 + 1 = 11 (Sum=1, Cout=1)
        A_tb <= '1'; B_tb <= '1'; Cin_tb <= '1';
        wait for 10 ns;

        -- Finalizar simulación
        wait;
    end process;

end Behavioral;
```