Documentación Ejercicio 4.

Para el diseño del módulo del Ripple Carry Adder (RCA) se diseña también un módulo de un sumador completo Full Adder, el cual se instancia dentro del módulo del RCA.

Para el diseño se consideró la tabla de verdad que modela el comportamiento de salida y entradas del Full Adder.

Tabla de verdad RCA				
Entradas			Salidas	
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

El modulo del sumador completo se describe a continuación.

```
module FullAdder(
    input a, b, cin,     // Entradas de un 1 bit del sumador completo
    output s, cout     // Salida de un 1 bit de la suma y acarreo
);
// Definición de la suma y el acarreo mediante compuertas lógicas
assign s = a ^ b ^ cin;
assign cout = (a & b) | (a & cin) | (b & cin);

endmodule
```

Instancia del módulo Full Adder en el módulo del RCA

```
logic [Ancho:0] local_Cout; // "salida" local para los carrys intermedios

assign local_Cout[0]= Cin; //el carry intermedio del bit 0 es el Cin
assign Cout = local_Cout[Ancho];

// Instancia del módulo de full adder
genvar i;
generate
    for (i = 0; i < Ancho; i = i + 1) begin : bit_
        FullAdder FA_ (
            .a(A[i]), //La entrada a del FA conectada con la entrada A del RCA
            .b(B[i]), //La entrada b del FA conectada con la entrada B del RCA
            .cin(local_Cout[i]), // La entrada cin del FA conectada con los carry intermedios del RCA (donde [0] es Cin -line15-)
            .s(S[i]), // La salida s del FA conectada con la salida S del RCA
            .cout(local_Cout[i+1]) // La salida cout del FA conectada con el último+1 carry intermedio del RCA
        );
    end
endgenerate

assign S[Ancho] = local Cout[Ancho]; //overflow
```

Se diseña además un Carry Lookahead Adder (CLA), el cual es mas eficiente que el RCA.

```
module cla #(
    parameter Ancho = 8
)(
    // Entradas del sumador
    input logic [Ancho-1:0] A,
    input logic [Ancho-1:0] B,
    input logic Cin,
    input logic clk, // Clock utilizado para el testeo de ruta crítica

    // Salidas del sumador
    output logic [Ancho:0] S,
    output logic Cout,
    output logic Overflow
);

logic [Ancho-1:0] G, P, C;
//logic [Ancho:0] C; // Cambiado a [Ancho:0] para incluir el acarreo final

// Inicialización para el bit 0
assign G[0] = A[0] & B[0];
assign P[0] = A[0] ^ B[0];
assign C[0] = Cin;

// Generación de G, P y C para bits 1 a Ancho
genvar i;
generate
    for (i = 1; i < Ancho; i = i + 1) begin : gen_for
        assign G[i] = A[i] & B[i];
        assign P[i] = A[i] ^ B[i];
        assign C[i] = G[i-1] | (P[i-1] & C[i-1]);
    end
    // Acarreo final
    assign C[Ancho] = G[Ancho-1] | (P[Ancho-1] & C[Ancho-1]);
endgenerate

// Asignación de valores de salida
assign S = A + B + Cin; // La suma será la suma de A, B y Cin
assign Cout = C[Ancho-1];
assign Overflow = C[Ancho] ^ C[Ancho-1]; // Overflow if MSB of carry is different from the next carry bit

endmodule
```

Constrains con frecuencia de reloj.

Frecuencia de 10 Mhz:

```
create_clock -period 10 -name clk -waveform {0 5} [get_ports clk]
set_max_delay -from [get_ports Cin] -to [get_ports Cout] 10
```

Frecuencia de 100 Mhz:

```
create_clock -period 100 -name clk -waveform {0 5} [get_ports clk]
set_max_delay -from [get_ports Cin] -to [get_ports Cout] 100
```