Examen

Arquitectura de Computadoras y Microcontroladores 1

Parcial 2

Alumno: Diego Andre Cuellar Butcher

Carrera: Ingeniería electrónica y telecomunicaciones

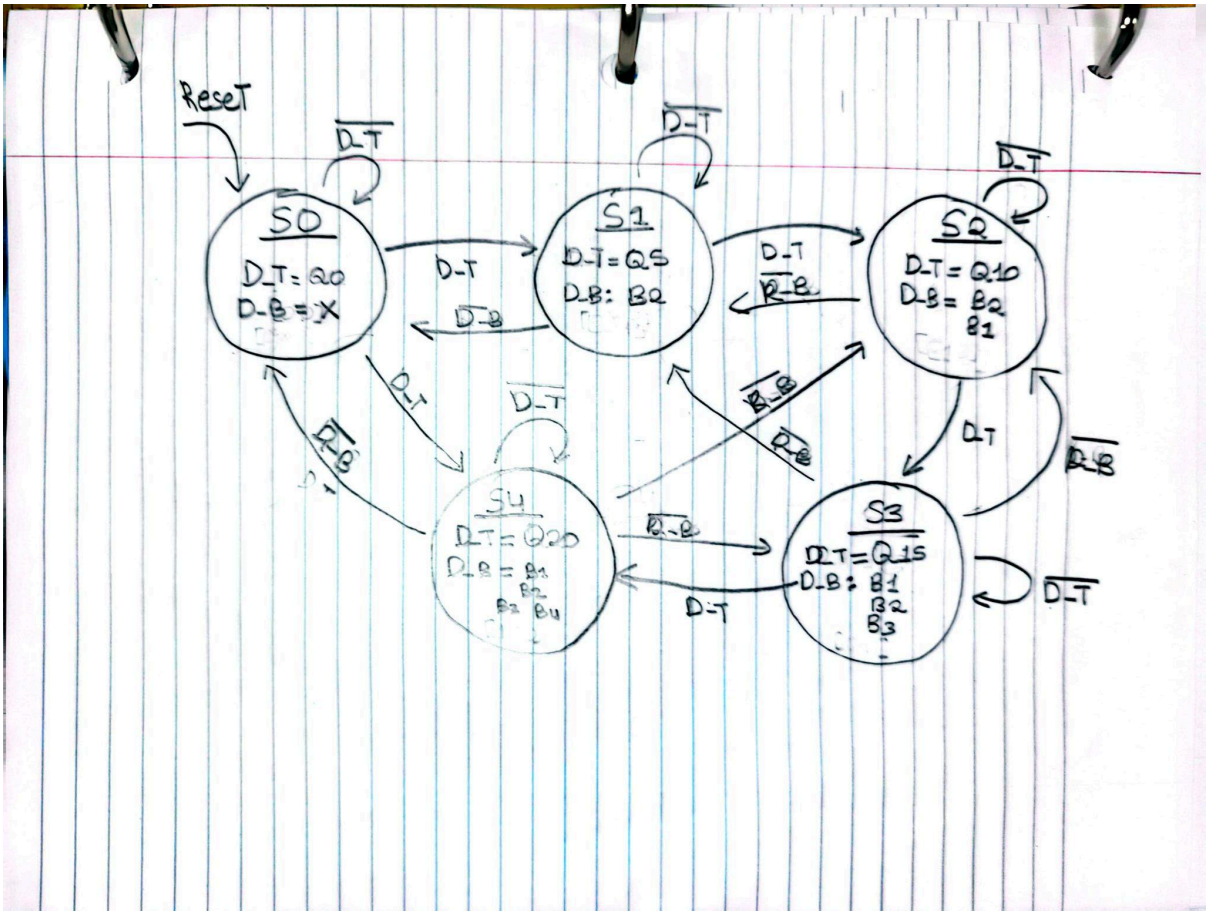
ID: 15735

Correo: dcuellar@unis.edu.gt

Guatemala
17 de septiembre, 2025

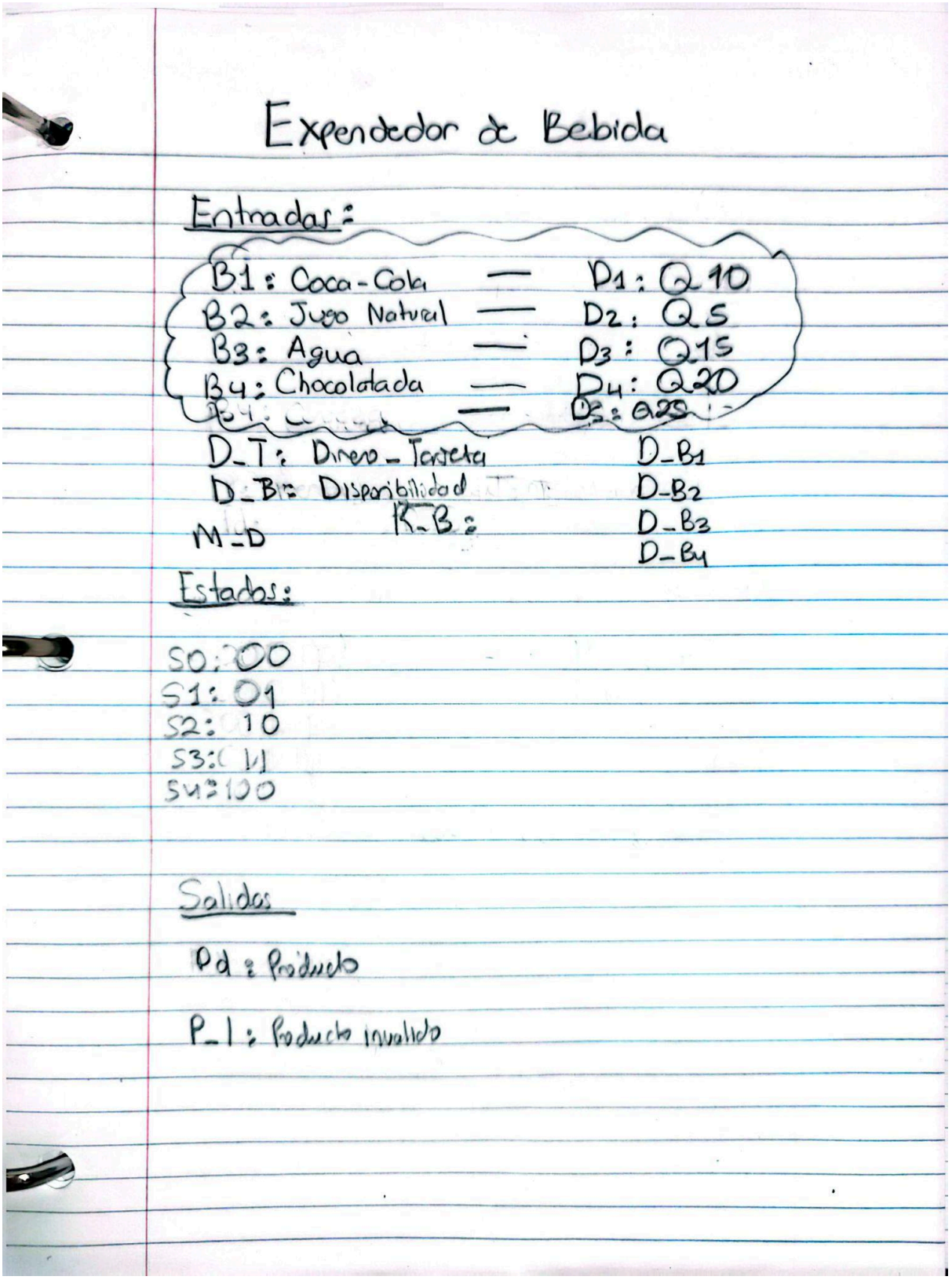
Serie 0:

-Tomando en cuenta que antes se hizo una máquina de estados finita (FSM), en donde ya se realizó sus entradas, salidas, estados y diagrama. Es lo que se muestra en las siguientes imágenes.



Se fueron utilizando las mismas tablas de verdad para sacer el resultado esperado del FSM de expendedora de bebidas.

Tablas: [Tablas FSM P1.xlsx]



-Pero ahora se va a implementar su funcionamiento en el código SystemVerilog en vivado.
En los videos de YouTube se explica el funcionamiento de su máquina:
Codigo en Vivado:FSM en System Verilog

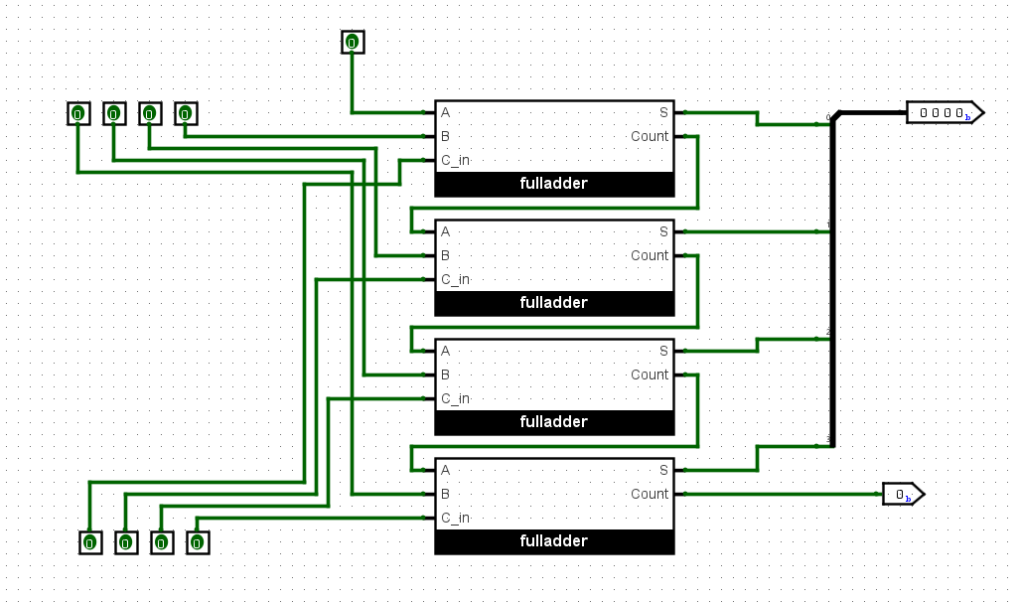
Serie 1:
En el siguiente enlace se presenta las simulaciones de los 3 sumadores solicitados:

- *Ripple-Carry Adder*
- *Carry-Lookahead Adder*
- *Prefix Adder*

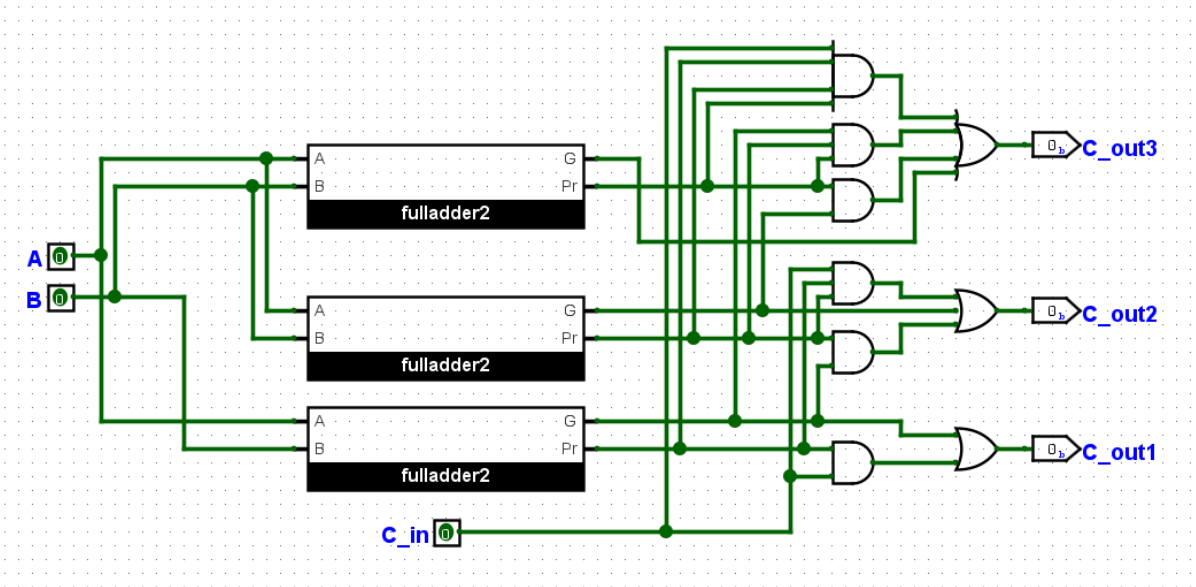
Simulaciones: P2_serie1.circ

Representación sobre los 3 sumadores en logisim:

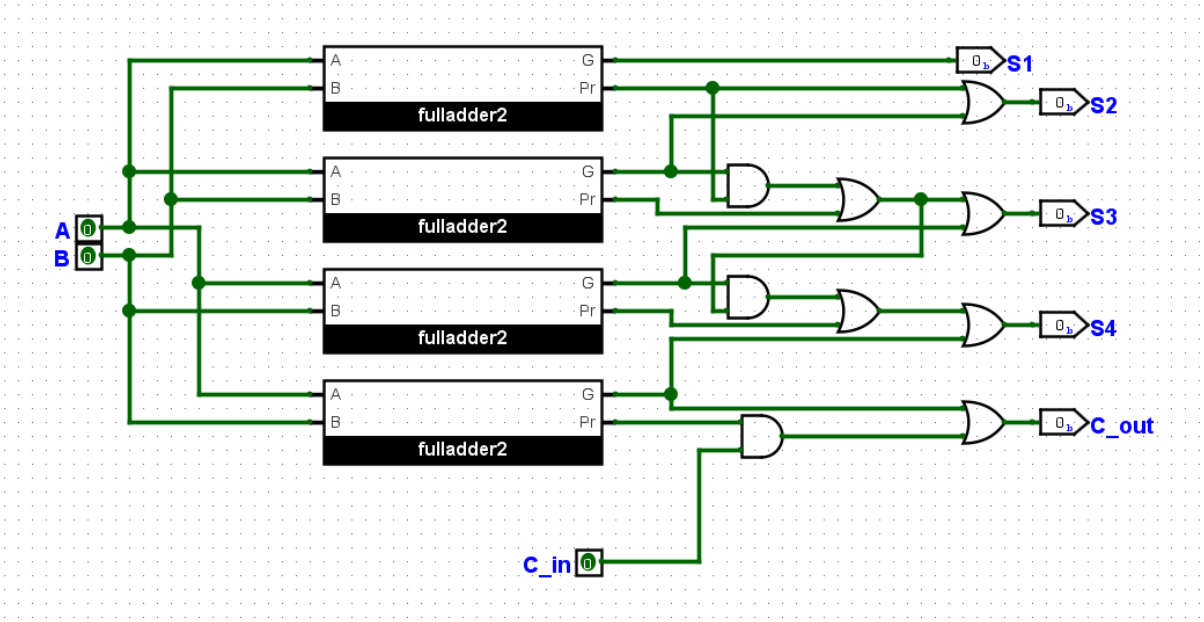
- *Ripple-Carry Adder*



- *Carry-Lookahead Adder*



- *Prefix Adder*



>Análisis de timing de cada sumador

1. ***Ripple***: Ahora que hemos visto en las simulaciones el funcionamiento de este adder, vamos a ver su tiempo de operación que se demora en su tiempo de respuesta.

   formula: t_ripple: Nt_FA

   >N: 5 bits
   >t_FA: 10 ns —> esto es más o menos o cercano a su tiempo tardío de respuesta del C_out del circuito.
   >t_ripple: (5)(10) = 50 ns

   El tiempo de demora del sumador sería de 50 ns.

2. ***Carry***: Ahora vamos con el análisis del carry adder.

   formula: t_CLA: t_pg + T_pg_block + (N/K - 1)t_AND_OR + kt_FA

   >N: 8 bits
   >T_pg_block: 3 ns —> Este vendría siendo el tiempo de repuestos del propagate “P” y del generate “G” en los niveles de los bloques.
   >t_pg: 2 ns este también vendría siendo el tiempo de respuesta de “P” y “G” aunque en este caso solo vendría siendo en solo en esos dos mismo valores. .
   > kt_FA: El tiempo de respuesta de su fulladder que vendría siendo en un aproximado de 5 ns

>t_AND_OR: Es el tiempo combinado de las compuertas AND y OR en el adder. Vendría siendo de un aproximado de 2 ns
>K: 4 bits, es los bits de cada bloque del adder.

Al realizar los cálculos de este análisis arroja que el total sería de 11 ns de tiempo de demora

3. ***Prefix***: Ahora continuamos con el tercero que sería el prefix adder.

Nota: Algo a notar es que en este también fue usando el mismo full adder que el sumador anterior asique algunos de los valores establecidos en la fórmula tienen el mismo significado.

formula: t_PA: t_pg + logN(t_pg_prefix) + t_XOR

>t_pg: 2 ns
>N: 4 bits
> t_XOR: 1 ns Este vendría siendo el tiempo de respuesta de la compuertas XOR. Aunque este vendría integrado al full adder de este adder.
>t_pg_prefix: 1 ns Tiempo estimado de operación de cada prefijo.
>logN: Profundidad de los bits del prefijo.

Después de realizar los cálculos, con los valores establecidos de la simulación arroja que el tiempo de demora es de: 3.60 ns

Resumen de análisis de tiempos:

- *Ripple-Carry Adder: 50 ns*
- *Carry-Lookahead Adder: 10 ns*
- *Prefix Adder: 3.60 ns*

Cuales usaría con énfasis en cantidad de compuertas y velocidad para aplicaciones:

- *Lentas con restricción de espacio y presupuesto*

  En este caso usaría el Ripple-Carry Adder. Esto es debido a que tiene un bajo consumo de datos y de una simplicidad estructural. Ya que usa la menor cantidad de compuertas en donde le es ideal para entornos limitados. A pesar de que su tiempo de propagación es lineal y más lento, hace que cada bit de respuesta espere el anterior, después de cada operación.

- Rápidas sin restricción de espacio y presupuesto

  En este caso utilizaría el Prefix Adder, debido a que su estructura y red jerárquica, permite operaciones de prefijo, en donde sus tiempos son algoritmos. Al consumir más datos y energía en otras palabras, se obtiene una velocidad de respuesta. Aunque utiliza más compuertas dependiendo de su capacidad de bits.

- Rápidas con restricción de espacio y presupuesto

En este caso usaría el Carry-Lookahead adder, debido a que este ofrece una velocidad ideal pero con restricción que son medidas balanceadas. Utiliza una cantidad necesaria de compuertas, lo que lo vuelve ideal para circuitos integrados como el FPAG.

Serie 3:

carne: 15735

Ultimo digito: 5

Penúltimo dígito: 3

(5*2)-3: 7 bits de buses de datos.

En una ALU hay cuatro operaciones establecidas:

Funciones	Bits	Operaciones
ADD	00	A + B
SUBTRACTION	01	A - B
AND	10	A & B
OR	11	A | B

Debido a que se requiere que agreguemos las siguientes operaciones: shift left, logical shift right, arithmetic shift right. Pero no debe excederse de 6 operaciones. Por lo que quedaría de los siguientes registros de operaciones.

Funciones	Bits	Operaciones
ADD	000	A + B
SUBTRACTION	001	A - B
AND	010	A & B
OR	011	A | B
shift left	100	A << B
shift right	101	A >> B

Banderas	Funcion
Zero	Cuando su valor es cero o no tiene ninguno
Carry	El sumador produce un carry out.
Negative	Produce un resultado negativo
Overflow	Produce un resultado desbordado

Aqui esta el enlace para el archivo de ALU: P2-Serie2

En el video se mostrará su funcionamiento.

Muestra del código:

```
module ALU (input  [6:0] A, B,
            input  [2:0] ALUControl,
            output reg [6:0] Result,
            output wire Zero, Carry, Overflow, Negative);


    reg [6:0] tmp;


    always @(*) begin
        case (ALUControl)
            3'b000: begin
                tmp = A + B;
                Result = tmp[6:0];
            end
            3'b001: begin
                tmp = A - B;
                Result = tmp[6:0];
            end
```

```
            3'b010: Result = A & B;
            3'b011: Result = A | B;
            3'b100: Result = A << 1;
            3'b101: Result = A >> 1;
            default: Result = 7'b0000000;
        endcase
    end

    assign Zero = (Result == 7'b0);
    assign Carry = tmp[7];
    assign Overflow = (ALUControl == 3'b000 || ALUControl == 3'b001) ?
                    ((A[6] == B[6]) && (Result[6] != A[6])) : 0;
    assign Negative = Result[6];
endmodule
```