UNIVERSIDAD DE COSTA RICA

ESCUELA DE INGENIERÍA ELÉCTRICA

Curso: Microelectrónica: Sistemas en Silicio
IE0411

Tarea Potencia - Reporte de resultados: sumadores

Estudiante
Brandon J. Esquivel Molina **B52571**
Profesor: Javier Pacheco Brito

Índice

Índice de figuras

Índice de cuadros

1. Introducción

Los circuitos integrados digitales CMOS VLSI actuales y muchos otros sistemas, realizados con tecnologías submicrométricas, alcanzan enormes velocidades de operación. En sus puertas lógicas, los retrasos de propagación se hacen cada vez más pequeños, a la vez que la enorme densidad de integración hace cada vez más complejos los caminos de interconexión. La potencia consumida toma ahora importancia entre estos aspectos principales, su manejo, cálculo y optimización resulta primordial para los sistemas actuales que requieren grandes tiempos de operación sin nueva carga.[1] En este reporte se muestran los resultados de un análisis de potencia propuesto desde un modulo especificado, donde se busca comprobar el efecto de usar distintos diseños con distintas compuertas, su consumo de potencia y sus retardos de propagación. Se utiliza una escala de referencia en 1 ns con una precisión en 1 ps en la implementación del código.

2. Diseño bajo análisis de Potencia y temporización

Se analizan tres sumadores diferentes para comparar sus consumos de potencia en diferentes cantidades de sumas, los cuales son el sumador lógico, el sumador look ahead y el sumador de rizado, éste ultimo es implementado mientras que los dos primeros son dados desde el enunciado.

2.1. Módulo sumador completo implementado

El módulo base implementado es el sumador completo, como se muestra a continuación su diagrama y su código:

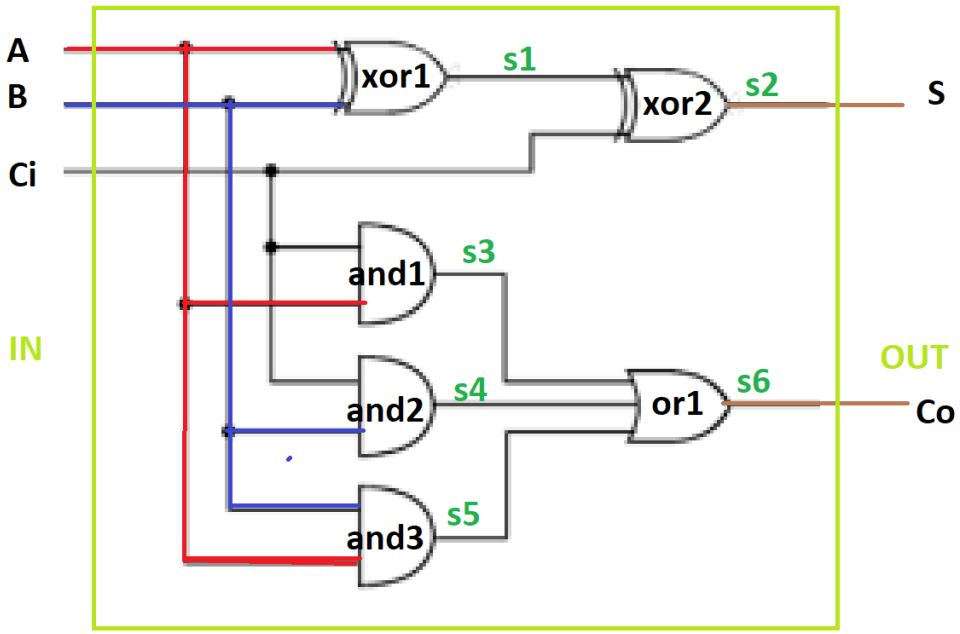

Figura 1: Diagrama del módulo sumador completo implementado

```
1
2  // ------------------------------------------------
3  // Sumador completo de 1-bit
4  // ------------------------------------------------
5  // Entradas: A, B, Cin
6  // Salida Cout y S
7  `timescale 1ns/1ps
8
9  module sumador_completo (a, b, ci, s, co);      // see
       complete adder diagram on docs
10   parameter
11     PwrC = 0;
12   input wire a,b,ci;
13   output reg s,co;
14   wire s1,s2,s3,s4,s5,s6;
15
16   xor2_p xor1(
17     //INPUTS
```

```
18      .c      (a),
19      .b      (b),
20      // OUTPUTS
21      .a      ( s1)
22    );
23
24      xor2_p xor2(
25      //INPUTS
26      .c      (s1),
27      .b      (ci),
28      // OUTPUTS
29      .a      (s2)
30    );
31
32  // carry logic
33    and2_p and1(
34      //INPUTS
35      .c      (ci),
36      .b      (a),
37      // OUTPUTS
38      .a      ( s3)
39    );
40
41      and2_p and2(
42      //INPUTS
43      .c      (ci),
44      .b      (b),
45      // OUTPUTS
46      .a      ( s4)
47    );
48
49      and2_p and3(
50      //INPUTS
51      .c      (a),
52      .b      (b),
53      // OUTPUTS
54      .a      ( s5)
55    );
56
57     or3_p or1(
```

```
58      //INPUTS
59      .b      (s3),
60      .c      (s4),
61      .d      (s5),
62      // OUTPUTS
63      .a      ( s6)
64    );
65
66
67    always@(*) begin
68      s = s2;
69      co = s6;
70    end
71
72  endmodule
```

2.2. Módulo sumador de rizado implementado

El modulo sumador de rizado diseñado utiliza el módulo anterior como base y su diagrama y código se muestran a continuación:

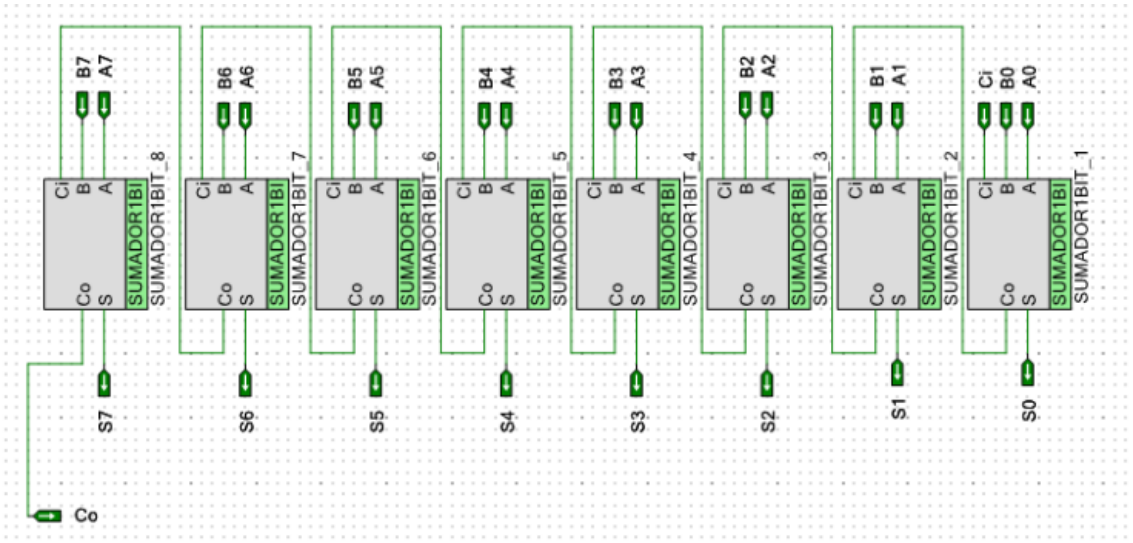

Figura 2: Diagrama del módulo sumador de rizado implementado

y su código:

```
`ifndef SUMADOR_RIZADO
`define SUMADOR_RIZADO
`timescale 1 ns / 1 ps

// ------------------------------------------------
//* Brandon Esquive Molina
//      brandon.esquivel@ucr.ac.cr
//      ripple adder module 8 bits
//
// ------------------------------------------------
// this module can be implemented more quickly with a
   generate block.

module SUM_RIZADO(a, b, ci, s, co);
  parameter
    PwrC = 0;
  input wire [7:0] a,b;
  input wire ci;
  output reg [7:0] s;
  output reg co;
  wire [7:0] carry, sn; //c0,c1,c2,c3,c4,c5,c6,c7,s0,s1,
     s2,s3,s4,s5,s6,s7;

  sumador_completo sum0(
    //inputs
    .a      (a[0]),
    .b      (b[0]),
    .ci     (ci),
    //outputs
    .s      (sn[0]),
    .co     (carry[0])
  );

  sumador_completo sum1(
    //inputs
    .a      (a[1]),
    .b      (b[1]),
    .ci     (carry[0]),
```

```
37      //outputs
38      .s      (sn[1]),
39      .co     (carry[1])
40    );
41
42    sumador_completo sum2(
43      //inputs
44      .a      (a[2]),
45      .b      (b[2]),
46      .ci     (carry[1]),
47      //outputs
48      .s      (   sn[2]),
49      .co     (carry[2])
50    );
51
52    sumador_completo sum3(
53      //inputs
54      .a      (a[3]),
55      .b      (b[3]),
56      .ci     (carry[2]),
57      //outputs
58      .s      (   sn[3]),
59      .co     (carry[3])
60    );
61
62    sumador_completo sum4(
63      //inputs
64      .a      (a[4]),
65      .b      (b[4]),
66      .ci     (carry[3]),
67      //outputs
68      .s      (   sn[4]),
69      .co     (carry[4])
70    );
71
72    sumador_completo sum5(
73      //inputs
74      .a      (a[5]),
75      .b      (b[5]),
76      .ci     (carry[4]),
```

```
77      //outputs
78      .s      (   sn[5]),
79      .co     (carry[5])
80    );
81
82    sumador_completo sum6(
83      //inputs
84      .a      (a[6]),
85      .b      (b[6]),
86      .ci     (carry[5]),
87      //outputs
88      .s      (   sn[6]),
89      .co     (carry[6])
90    );
91
92    sumador_completo sum7(
93      //inputs
94      .a      (a[7]),
95      .b      (b[7]),
96      .ci     (carry[6]),
97      //outputs
98      .s      (   sn[7]),
99      .co     (carry[7])
100   );
101
102   always@(*) begin
103     s = sn[7:0];
104     co = carry[7];
105   end
106
107 endmodule
108 `endif
```

Todos los códigos se encuentran comentados en el repositorio del proyecto para consulta.

2.3. Pruebas realizadas

Se utilizaron diferentes semillas para generaciones aleatorias, además de un número dado de sumas totales para los tres sumadores, con esto se midió

la potencia consumida en las transiciones de cada uno y sus retardos. Las cantidades de sumas utilizadas son 500, 1000, 2000 y 5000 sumas, cada una con tres semillas diferentes. Luego para los retardos se usaron 3 pares de operandos para comparar los resultados, estos correspondieron a (A,B) = ($00,$00), ($00,$01) y ($FF,$01)

Luego se Modificó el tiempo de retardo en el archivo definiciones.v usando el mismo valor del inversor para el resto de compuertas y nuevamente se obtuvieron los resultados de la primera parte. Finalmente se Modificó el diseño de la figura 1 y en lugar de utilizar dos compuertas XOR de dos entradas para la lógica de la salida S se utiliza una sola compuerta XOR de tres entradas, corriendo de nuevo con los retardos originales para las compuertas.

3. Resultados encontrados y análisis

3.1. Comportamiento adecuado del sumador de Rizado

Como se solicitó se muestra a continuación el resultado de operacion correcto del modulo sumador de rizado implementado:

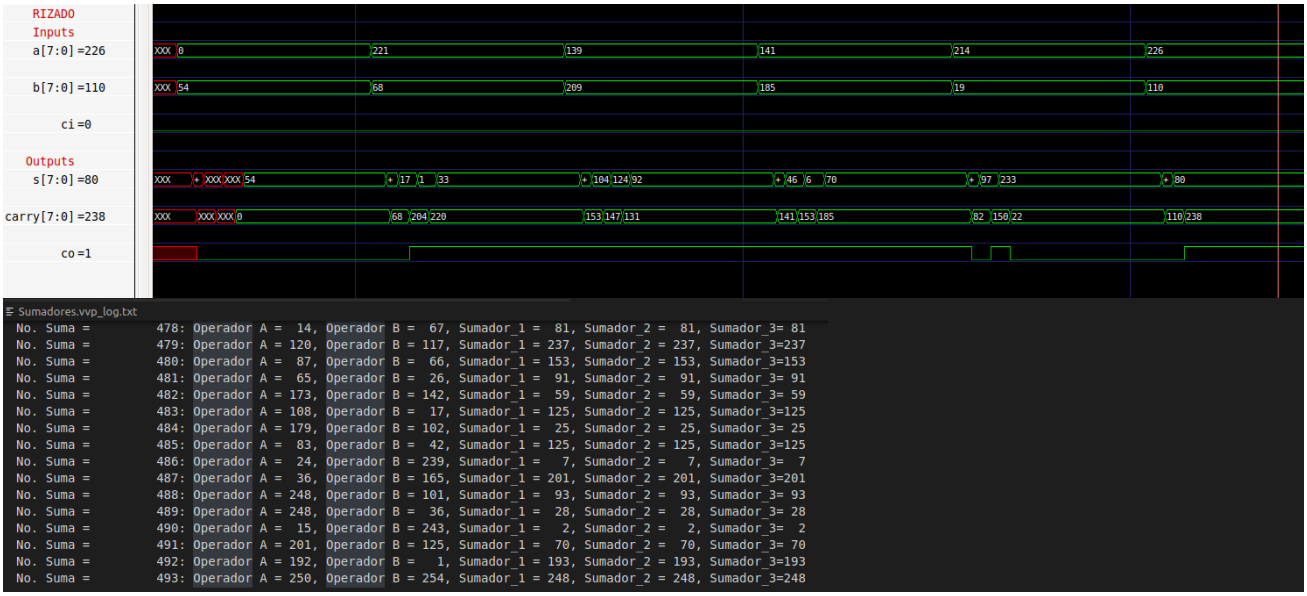

Figura 3: Resultado de simulación para el sumador de rizado implementado

Se puede apreciar que su funcionamiento es el adecuado, pues genera los valores correctos, los cuales son los mismos a los módulos sumadores dados.

3.2. Semillas y sumas

Se realizan las distintas pruebas con semillas y cantidades de sumas, para obtener el consumo de potencia de los tres sumadores. En este primer caso, los resultado se resumen en la siguiente tabla:

Semilla	Potencia Rizado	Potencia lógico	Potencia look
Para 500 sumas			
1117544411	495240	571560	492000
95644445	516480	595380	518940
-311621822	501240	585060	509520
Para 1000 sumas			
1191698099	1005420	1156560	1002360
271924533	1029780	1184820	1037400
119974266	1022340	1174680	1023360
Para 2000 sumas			
1298874090	2039100	2338920	2039340
749573733	2037060	2362860	2024640
-329139869	1997400	2309640	2012160
Para 5000 sumas			
-745059851	5051760	5834880	5019060
-994496141	5039820	5805780	5015220
2026014522	5033220	5820120	5029320

Cuadro 1: Resultados de potencia de sumadores por semilla y cantidad de sumas

Se puede apreciar como era de esperar que la potencia es directamente proporcional al numero de operaciones realizadas, esto es más específicamente, a las transiciones realizadas de cero a uno (medición realizada). Por otro lado se puede ver que para menor cantidad de sumas, el sumador de rizado presenta menor consumo de potencia, muy cercano al sumador look ahead, el cual mantiene el menor consumo para mayores cantidades de sumas. Por su lado el sumador lógico mantiene el mayor consumo de potencia para los casos.

3.3. Retardos

al realizar las pruebas de retardo se encontraron los siguientes resultados:

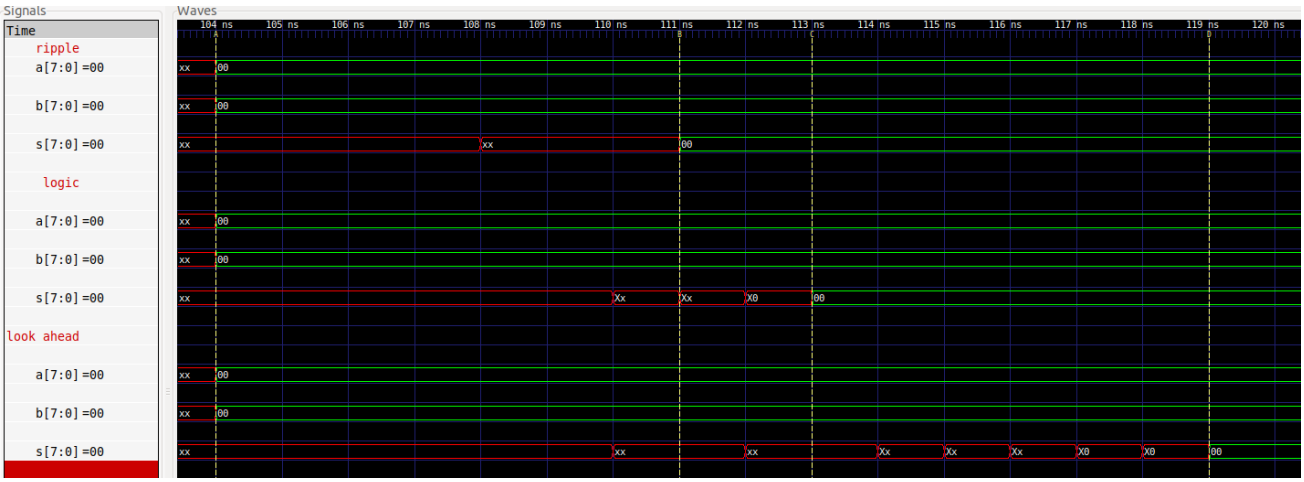

Figura 4: Resultado de simulación para retardos con A=0, B=0

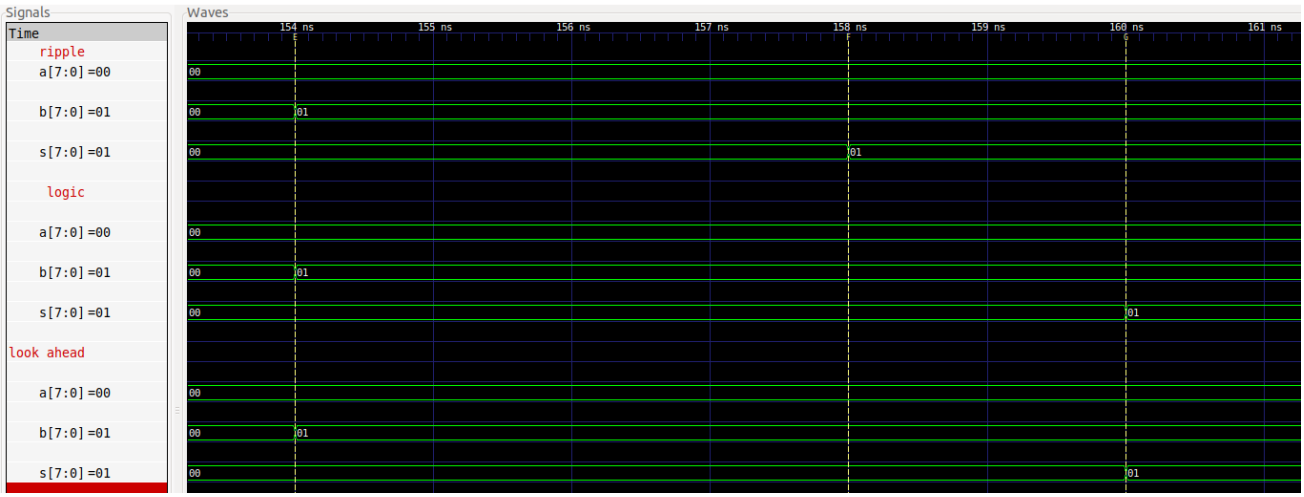

Figura 5: Resultado de simulación para retardos con A=0, B=1

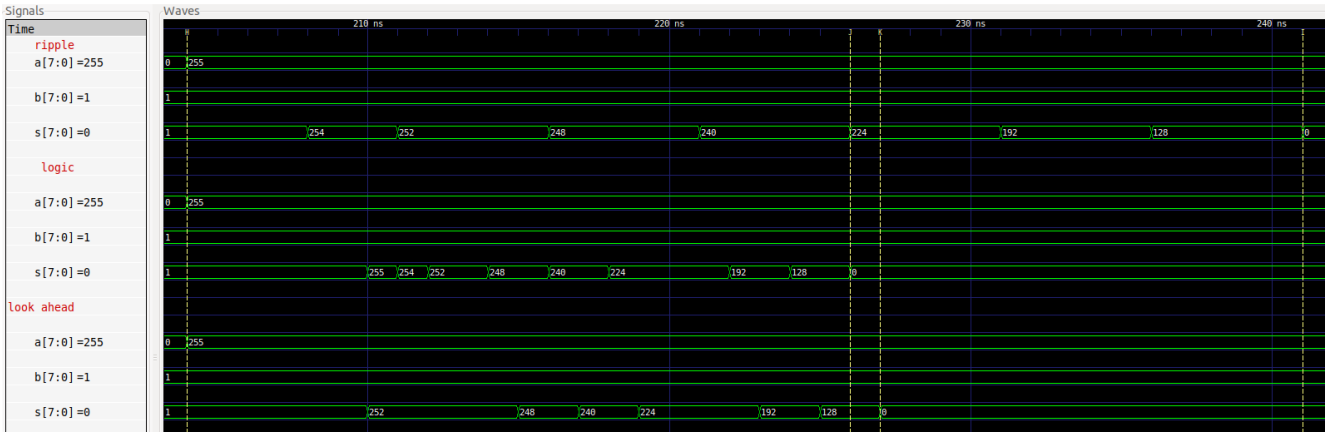

Figura 6: Resultado de simulación para retardos con A=FF, B=01

Se puede apreciar que para la primer suma en la figura 4 el sumador de rizado es el más rápido con un retardo de propagación de 7ns mientras que le sigue el sumador lógico con un retardo de 9ns y por último el sumador look ahead con 15ns. Para la segunda suma, en la figura 5 se aprecia que nuevamente el sumador de rizado posee el menor retardo de propagación siendo de 4ns, seguido por los dos restantes con 6ns ambos. Finalmente para la última suma de la figura 6 se ve que el sumador lógico es el de menor retardo de propagación con 22 ns, seguido del sumador look ahead con 23ns y por último el sumador de rizado con 37ns. De estos resultados se puede decir que el sumador ripple funciona de forma más rápida para sumas con valores cercanos, es decir con menores transiciones de bits, mientras que para mayores números, cambios o transiciones, resulta con mayor tiempo de propagación, por su lado los sumadores restantes muestran un comportamiento similar en términos de retardo para más transiciones, y el lógico representa un punto medio para transiciones menores.

En el repositorio se encuentra la carpeta log donde encontrará el archivo .log que muestra todos los puntos de simulación de importancia.

3.4. Modificando tiempo de retardo

se Modificó el tiempo de retardo en el archivo definiciones.v usando el mismo valor del inversor para el resto de compuertas y nuevamente se obtuvieron los resultados de la primera parte:

Figura 7: Cambio realizado a definiciones.v para mantener un retardo de inversor (1)

Los nuevos retardos se muestran en las siguientes figuras:

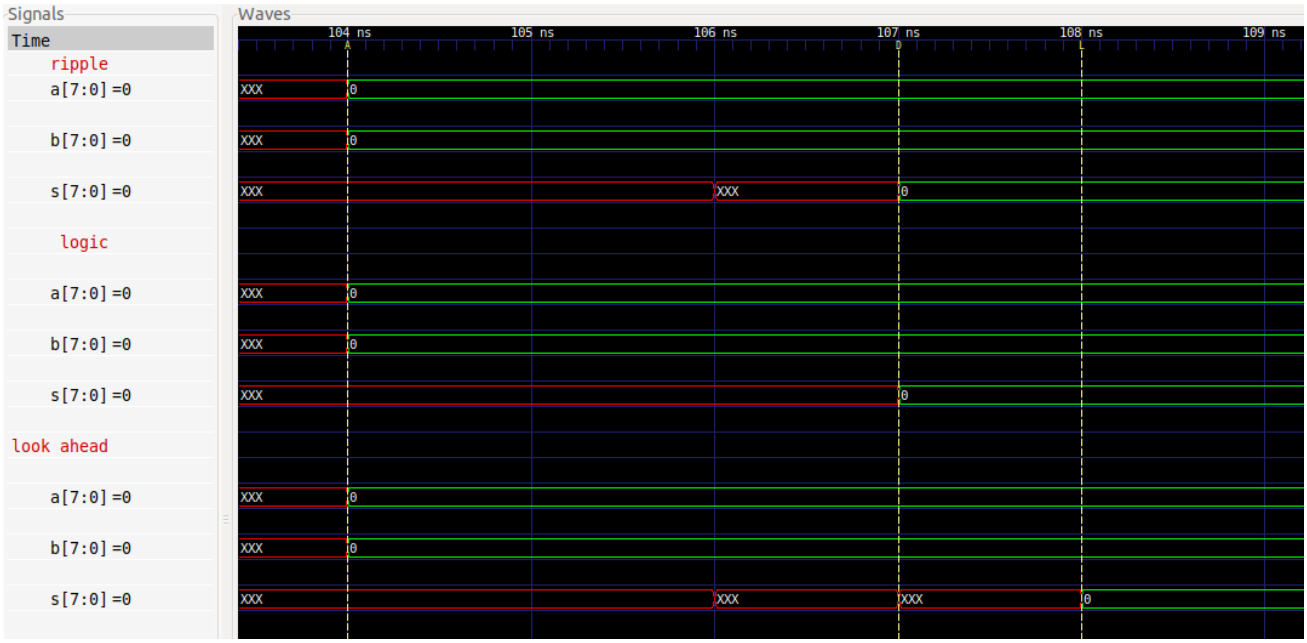

Figura 8: Resultado de simulación para retardos con A=0, B=0 con modificación.

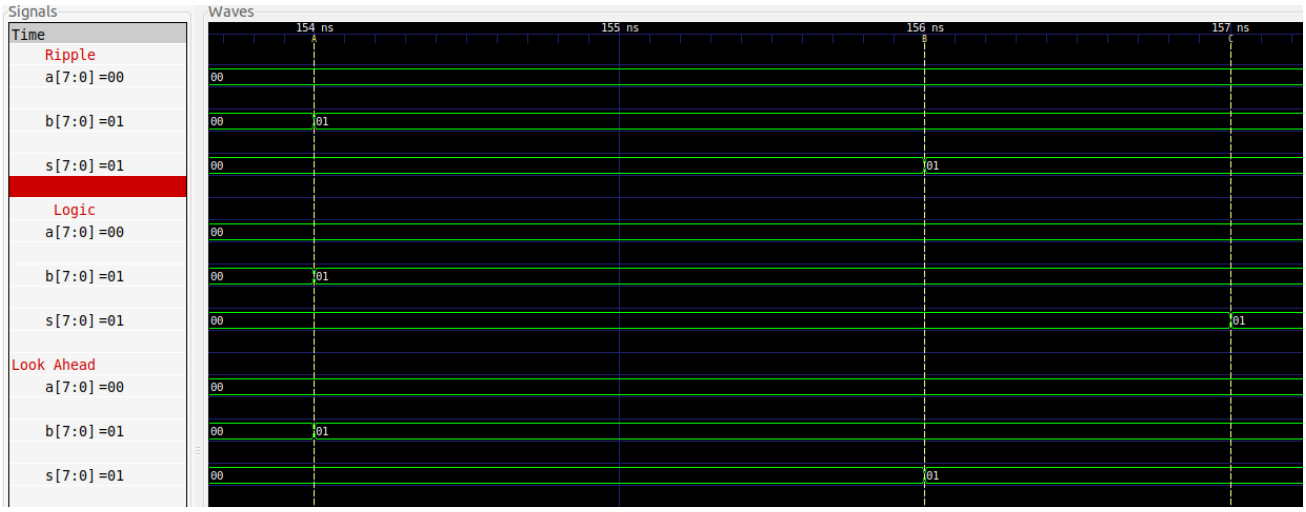

Figura 9: Resultado de simulación para retardos con A=0, B=1 con modificación

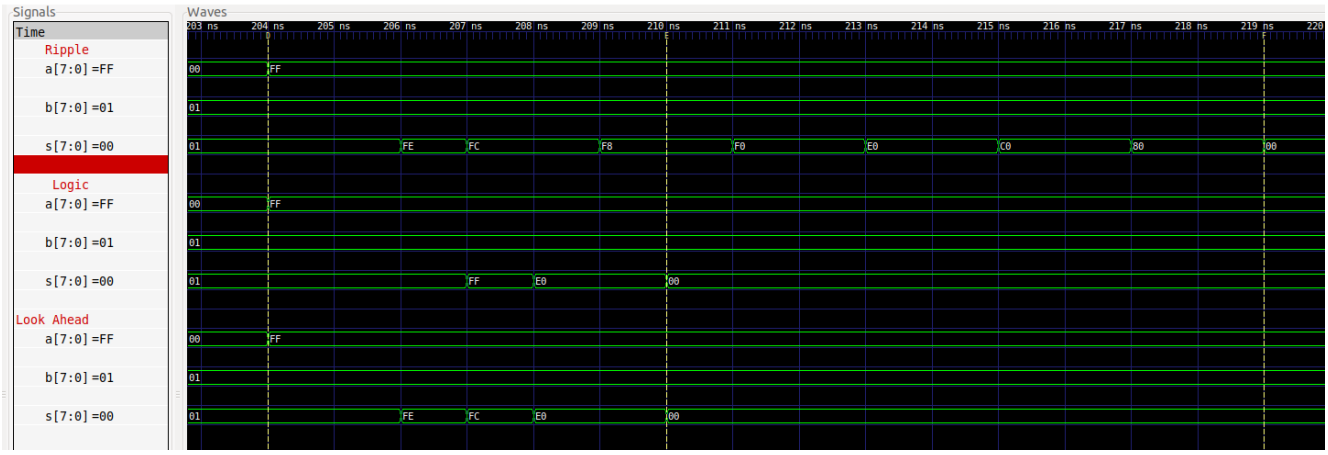

Figura 10: Resultado de simulación para retardos con A=FF, B=01 con modificación

Se puede apreciar claramente que los retardos disminuyeron en comparación al punto anterior, como era de esperarse al modificar el valor hacia el retardo de un inversos (unidad). En el primer caso, el retardo fue de 3ns para sumador rizado y lógico, y de 4ns para look ahead(figura 8), luego en la figura 9 se aprecia que para rizado y look ahead el retardo es de 2ns, mientras que

fue de 3ns para el sumador lógico. Finalmente en la figura 10 se muestra un retardo de 6ns para el sumador lógico y look ahead, mientras que el rizado mantiene 15ns. Al disminuir los retardos, los tiempos de propagación se ven disminuidos, y los sumadores se mantiene cercanos, aunque se mantiene el mismo comportamiento descrito en en la sección anterior

3.5. Cambio en el diseño

Se modificó el diseño de la figura 1 para obtener el siguiente diseño:

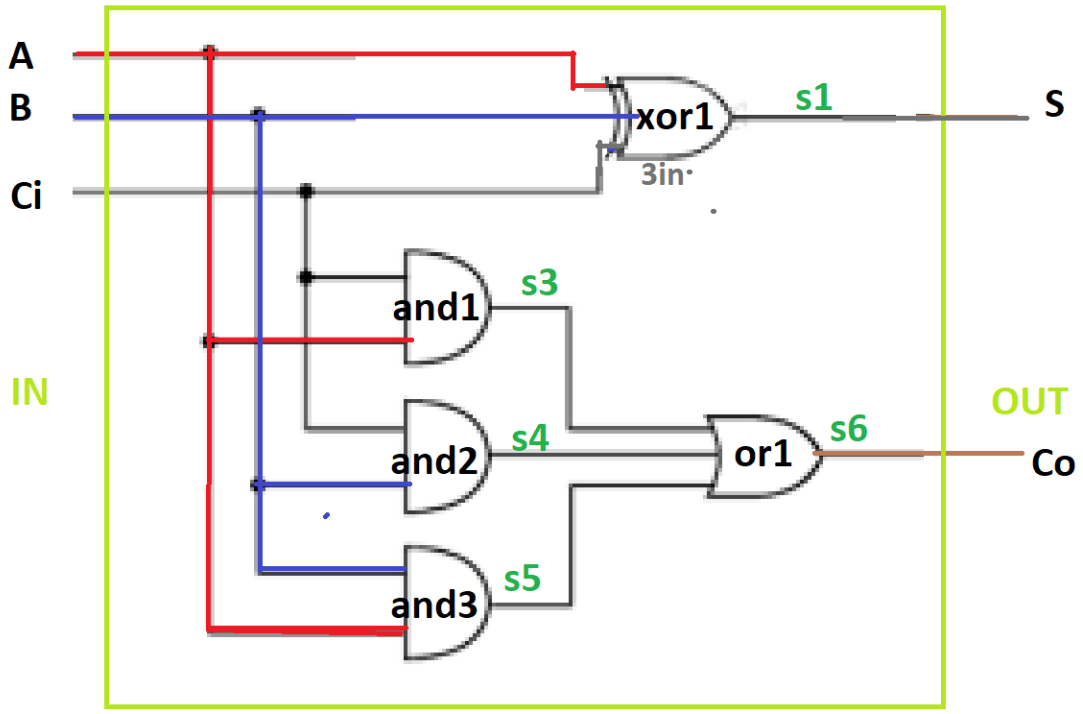

Figura 11: Diagrama del nuevo diseño

En lugar de utilizar dos compuertas XOR de dos entradas para la lógica de la salida S utilice una sola compuerta XOR de tres entradas. Se volvieron a realizar los cálculos de retardos para obtener los siguientes resultados:

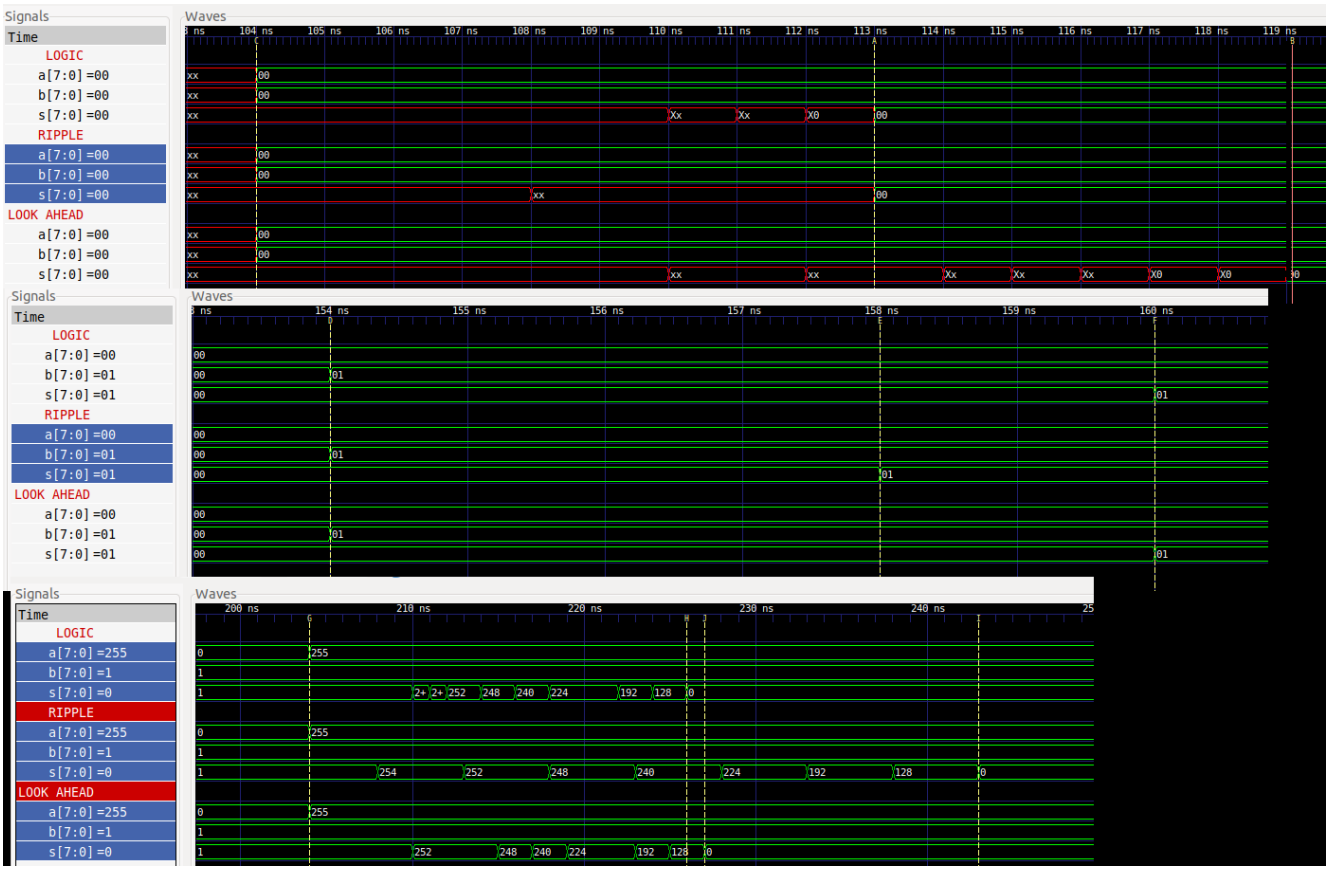

Figura 12: Resultados de simulación con nuevo diseño, retardos de propagación.

Se puede apreciar que para el primer caso tanto rizado como lógico mantienen un retardo de 9ns, mientras que look ahead presenta 15ns, luego en el segundo caso, lógico y look ahead presentan 6ns mientras que rizado alcanza solo 4ns, finalmente se puede ver que para el caso de mayor transiciones rizado llega a 39ns, siendo el mayor retardo, mientras que el sumador lógico alcanza 22ns y look ahead 23ns.

3.6. Tabla comparativa

Se resumen ahora los resultado anteriormente analizados en la siguiente tabla:

Prueba	**Diseño inicial, pruebas aisladas de sumas generales**		
Transición (A,B)	**Sumador Rizado**	**Sumador lógico**	**Sumador look ahead**
(00, 00)	7ns	9ns	15ns
(00, 01)	4ns	6ns	6ns
(FF, 01)	37ns	22ns	23ns
Prueba	**Modificación tiempo de retardo a 1 (del inversor)**		
(00, 00)	3ns	3ns	4ns
(00, 01)	2ns	3ns	2ns
(FF, 01)	15ns	6ns	6ns
Prueba	**Nuevo diseño con XOR de 3 entradas y retardos iniciales**		
(00, 00)	9ns	9ns	15ns
(00, 01)	4ns	6ns	6ns
(FF, 01)	39ns	22ns	23ns

Cuadro 2: cuadro comparativo con los resultados de los casos de análisis

Una vez recopilados los casos se puede ver primeramente que el nuevo diseño mantiene mayores tiempos de propagación que el original, esto es lo esperado al cambiar a una Xor de 3 entradas. Al pasar de un inversor a una puerta booleana de dos o más entradas, los tiempos de propagación aumentan en aquellos procesos (carga o descarga) en que la conducción se produce a través de varios transistores en serie; es un efecto de suma de sus resistencias (se suma la longitud de sus transistores). Por otro lado la reducción de los retardos hacia 1 en el archivo definiciones.v reduce los tiempos de propagación a menos de la mitad en los sumadores, como es de esperarse al acercarse a la idealidad.

4. Enlace del repositorio

`https://github.com/brandonEsquivel/Power_dissipation_analysis`

5. Observaciones y Recomendaciones

Los procesos de medición de retardos y potencia son esenciales en los sistemas CMOS, un buen diseño puede llevar a excelentes resultados de eficiencia y velocidad. Se nota entonces la importancia de la temporización de circuitos digitales y módulos, pues su análisis es fundamental para el correcto funcionamiento de los sistemas. El conocimiento claro y conciso del proceso de diseño de sumadores ayudar a mejorar la capacidad de análisis de los resultados y la solución de posibles errores presentes en la simulación y compilación. Un bloque generate puede facilitar la generación del sumador de rizado, haciéndolo de forma automática y para N-bits deseados. Además de la disipación de potencia de tipo capacitivo (principalmente la capacidad de entrada de los transistores MOS) existe otro efecto dinámico debido a que en la conmutación, durante un breve instante de tiempo, conducen ambos transistores PMOS y NMOS, dando lugar a un estrecho "pico de intensidad"; tal efecto resulta despreciable frente al anterior siempre que la conmutación sea adecuadamente rápida (tiempos de conmutación inferiores a 1 ns). Sobre la potencia en compuertas, para cada valor booleano en una de las entradas, uno de sus transistores se encontrará en corte y el otro conducirá: todo camino de conducción entre los dos terminales de alimentación (VCC y 0 V) incluye siempre un transistor en corte, por lo cual el consumo en reposo es nulo. En cambio, sí que hay consumo dinámico originado por la carga o descaga de las diversas capacitancias propias de los transistores en la conmutación, es decir en las transiciones y dicho consumo es proporcional a la frecuencia de conmutación.

Referencias

[1] A. Acosta and A. Jiménez, *Temporización en Circuitos Integrados Digitales CMOS*. ACCESO RÁPIDO, Marcombo, S.A., 2000.