

Trabajo Práctico 9 - Taller de Arquitectura

Multiplicador de 4 bits



Ringuelet Pedro

02960/7

Índice

1. Introducción	3
2. Desarrollo	3
2.1 Acumulador	3
2.2 Multiplicador de 4 bits	4
3. Testbench y Validación.....	6
3.1 Selección de Datos de Prueba	6
3.2 Simulación y Verificación	7
4. Análisis de Tiempos	8
4.1 Caso 1(9×2)	8
4.2 Caso 2(2×9)	9
4.3 Cálculo de XX	10
5. Conclusión	10

1. Introducción

Este trabajo práctico tiene como objetivo el desarrollo e implementación de un multiplicador de 4 bits sin signo, utilizando componentes previamente diseñados en los Trabajos Prácticos 2 a 8. La implementación del multiplicador se realiza mediante una arquitectura controlada por una máquina de estados, que gestiona los pasos de carga, desplazamiento y acumulación necesarios para completar la operación de multiplicación. Para validar el funcionamiento del multiplicador, se construyó un testbench que simula la multiplicación de dos valores basados en el número de legajo del alumno. Además, se realiza un análisis de los tiempos de operación del circuito, considerando un reloj de **XX MHz**, y se comparan los tiempos de cálculo para $A \times B$ y $B \times A$ con el fin de verificar si son equivalentes.

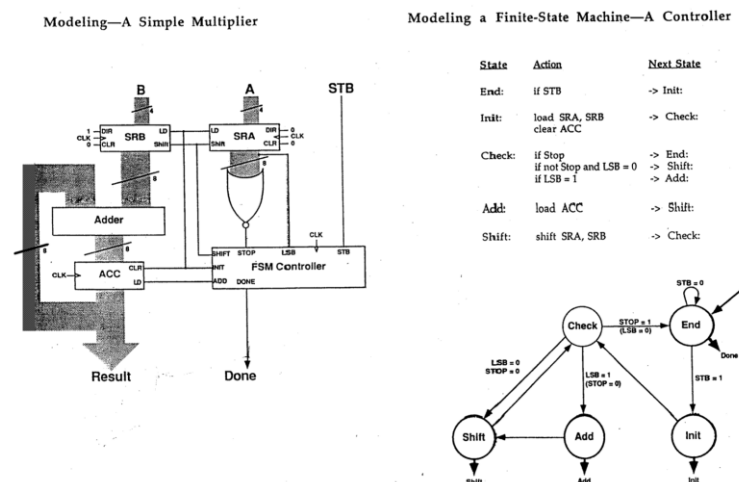


Figura 1: Multiplicador y FSM

2. Desarrollo

2.1 Acumulador

El Acumulador es un componente fundamental en la operación de multiplicación, ya que almacena el resultado parcial en cada ciclo y permite acumular la suma de los productos parciales. Su configuración incluye:

- **D:** Entrada de datos que representa el valor de la suma parcial generada por el sumador **Adder8**.
- **Clk:** Señal de reloj que sincroniza el proceso de carga y reinicio del acumulador.

- **Clr:** Señal de limpieza que restablece el valor del acumulador a 0 cuando se activa, usada al inicio de la multiplicación.
- **LD:** Señal de carga que habilita la actualización del valor en el acumulador con el valor de D.
- **Q:** Salida del acumulador que almacena el resultado parcial en cada ciclo y se actualiza en el siguiente ciclo de reloj, acumulando hasta que la operación de multiplicación esté completa.

```

1  entity Accumulator is
2      port (
3          D      : in  Bit_Vector(7 downto 0);
4          Clk     : in  Bit;
5          Clr     : in  Bit;
6          LD      : in  Bit;
7          Q       : out Bit_Vector(7 downto 0)
8      );
9  end Accumulator;
10
11 architecture Behave of Accumulator is
12 begin
13     acc: process(Clk)
14     begin
15         if Clk'Event and Clk = '1' then
16             if Clr = '1' then
17                 Q <= (others => '0'); -- Clear
18             elsif LD = '1' then
19                 Q <= D; -- Load
20             end if;
21         end if;
22     end process;
23 end Behave;

```

Figura 2: Código del Acumulador

Este mecanismo permite que el acumulador actualice el resultado de forma acumulativa en cada ciclo de multiplicación, hasta que se completa la operación y el resultado final está disponible.

2.2 Multiplicador de 4 bits

El multiplicador de 4 bits realiza la multiplicación entre dos números binarios sin signo de 4 bits, produciendo un resultado de 8 bits. Este diseño sigue una arquitectura controlada por una máquina de estados que coordina el flujo de la operación de multiplicación. Los componentes clave del multiplicador son:

1. **SR_A (Shift Register A):** Registro de desplazamiento que toma el operando A_8 y lo desplaza hacia la derecha en cada ciclo. Este desplazamiento permite que el

bit menos significativo (LSB) sea evaluado en cada ciclo, determinando si se activa la suma con **Adder8**.

2. **SR_B (Shift Register B)**: Registro de desplazamiento que toma el operando B₄ y lo desplaza hacia la izquierda en cada ciclo.
3. **Adder8**: Sumador de 8 bits que se usa para sumar el contenido del acumulador con el valor desplazado de **SR_B** cuando el LSB en **SR_A** es 1.
4. **Controller FSM**: Máquina de estados finitos que controla la operación del multiplicador. Los estados de la máquina son:
 - **Init**: Inicializa los registros de desplazamiento y el acumulador.
 - **Check**: Evalúa el LSB de **SR_A**; si es 1, activa la señal de suma (**Add**); si es 0, pasa al siguiente estado de desplazamiento.
 - **Add**: Suma el valor en el acumulador con el valor de **SR_B** desplazado.
 - **Shift**: Desplaza los registros **SR_A** a la derecha y **SR_B** a la izquierda.
 - **End**: indica la finalización de la operación de multiplicación y activa la señal Done.
 - **Done**: se activa una vez que el proceso de multiplicación ha concluido, permitiendo verificar que el resultado en **Result** es el final.
5. **Accumulator**: Componente acumulador, que recibe el valor parcial sumado y lo almacena para la siguiente iteración. Este acumulador se encarga de llevar el resultado parcial en cada ciclo hasta que la multiplicación está completa.

```

1  entity Mul4 is
2      port (
3          A_4, B_4: in  Bit_Vector(3 downto 0);
4          STB, CLK: in  Bit;
5          Result: out Bit_Vector(7 downto 0);
6          Done: out Bit
7      );
8  end Mul4;
9
10 architecture mul of Mul4 is
11
12     -- Componentes
13     component ShiftN
14         port (CLK, CLR, LD, SH, DIR: in Bit; D: in Bit_Vector; Q: out Bit_Vector);
15     end component;
16
17     component Adder8
18         port (A, B: in Bit_Vector(7 downto 0); Cin: in Bit; Cout: out Bit; Sum: out Bit_Vector(7 downto 0));
19     end component;
20
21     component Controller
22         port (STB, CLK, LSB, Stop: in Bit; Init, Shift, Add, Done: out Bit);
23     end component;
24
25     component Accumulator
26         port (D: in Bit_Vector(7 downto 0); Clk, Clr, LD: in Bit; Q: out Bit_Vector(7 downto 0));
27     end component;
28
29     -- Señales internas
30     signal QA, QB, Sum, Res, A_8, B_8: Bit_Vector(7 downto 0);
31     signal Init, Shift, Add, Stop: Bit;
32
33     -- Función auxiliar que replica la compuerta NOR
34     function compuerta_NOR(input_vector: Bit_Vector) return Bit is
35     begin
36         if input_vector = "00000000" then
37             return '1';
38         else
39             return '0';
40         end if;
41     end compuerta_NOR;
42
43 begin
44     -- Extender los operandos A y B a 8 bits (rellenando con ceros a la izquierda)
45     A_8 <= "0000" & A_4;
46     B_8 <= "0000" & B_4;
47
48     -- Instancia de Shift Register A (SRA)
49     SR_A: ShiftN port map(CLK, '0', Init, Shift, '0', A_8, QA);
50
51     -- Instancia de Shift Register B (SRB)
52     SR_B: ShiftN port map(CLK, '0', Init, Shift, '1', B_8, QB);
53
54     -- Generación de la señal de Stop usando la función compuerta_NOR
55     Stop <= compuerta_NOR(QA);
56
57     -- Instancia del Adder8
58     Adder: Adder8 port map(Res, QB, '0', open, Sum);
59
60     -- Instancia del Acumulador
61     ACC: Accumulator port map(Sum, CLK, Init, Add, Res);
62
63     -- Instancia de la FSM Controller
64     FSM: Controller port map(STB, CLK, QA(0), Stop, Init, Shift, Add, Done);
65
66     -- Asignación de la salida del resultado
67     Result <= Res;
68
69 end mul;
70
71

```

Figura 3: Código del Multiplicador

3. Testbench y Validación

3.1 Selección de Datos de Prueba

Para la prueba del multiplicador, se seleccionaron los siguientes valores de acuerdo al número de legajo 0290/7:

- **A:** Dígito de mayor valor del legajo (9).
- **B:** Dígito de menor valor del legajo distinto de 0 (2).

3.2 Simulación y Verificación

La simulación se realizó para verificar que el multiplicador produce el resultado correcto para los valores seleccionados. Se adjunta la imagen de la simulación que muestra el funcionamiento del multiplicador con las entradas A = 9 y B = 2. El resultado esperado de la multiplicación es $9 \times 2 = 18$. La simulación muestra que el circuito genera el resultado correcto, y la señal Done se activa al completar la operación, validando el funcionamiento del multiplicador.

```
1 use work.Utils.all; -- Importa el paquete Utils con Clock y Convert
2
3 entity Test_Mul4 is
4 end Test_Mul4;
5
6 architecture Test of Test_Mul4 is
7     -- Declaración del componente a testear
8     component Mul4
9     port (
10         A_4, B_4: in  Bit_Vector(3 downto 0);
11         STB, CLK: in  Bit;
12         Result: out Bit_Vector(7 downto 0);
13         Done: out Bit
14     );
15 end component;
16
17 -- Señales de prueba
18 signal A_4, B_4 : Bit_Vector(3 downto 0);
19 signal STB, CLK : Bit;
20 signal Result   : Bit_Vector(7 downto 0);
21 signal Done     : Bit;
22
23 begin
24     -- Generación del reloj
25     Clock(CLK, 6 ns, 6 ns); -- Procedimiento que genera la señal de reloj
26
27     -- Instancia del multiplicador usando mapeo posicional
28     mul: Mul4 port map(A_4, B_4, STB, CLK, Result, Done);
29
30     -- Inicialización
31     A_4 <= Convert(9,4); -- Convierte el número 9 en un vector de 4 bits
32     B_4 <= Convert(2,4); -- Convierte el número 2 en un vector de 4 bits
33
34     -- Inicio de la multiplicación
35     STB <= '0', '1' after 20 ns, '0' after 40 ns;
36
37     assert (Done = '0')| report "Simulacion finalizada con exito. " severity Note;
38
39 end Test;
```

Figura 4: Código del Testbench

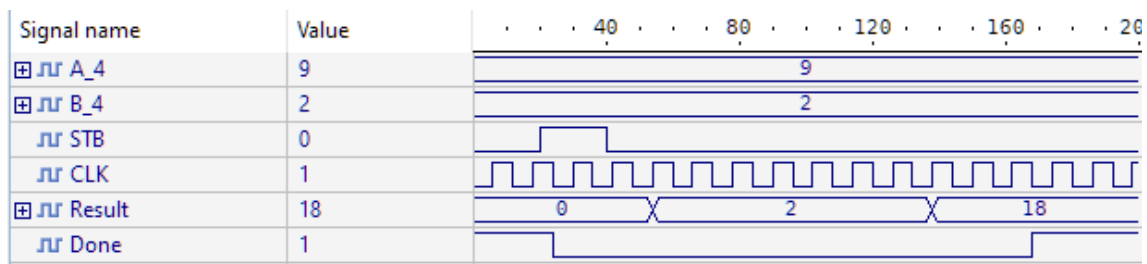


Figura 5: Waveform de 9x2

```

Console
# EXECUTION:: NOTE : Simulacion finalizada con exito.
# EXECUTION:: Time: 168 ns, Iteration: 2, Instance: /Test_Mul4, Process: line_37.
# KERNEL: stopped at time: 200 ns
endsim
# VSIM: Simulation has finished.

```

Figura 6: Visualización del Note en la consola (9x2)

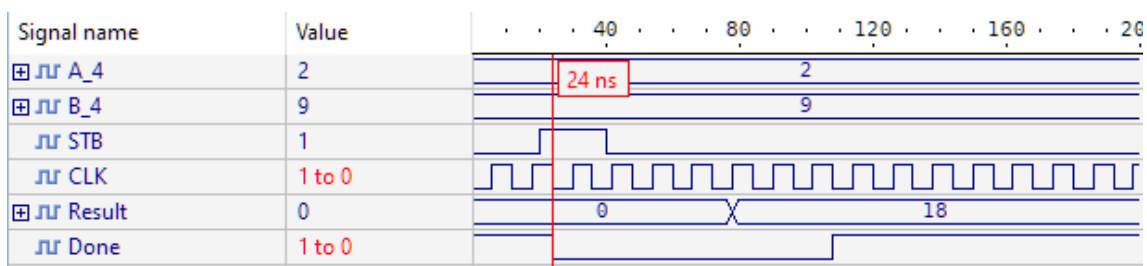


Figura 7: Instante en el que comienza la multiplicación

4. Análisis de Tiempos

Para un reloj de **87 MHz**, el período del ciclo de reloj es aproximadamente 12 ns (calculado como $1/(87\text{MHz})$ convertido a nanosegundos). El tiempo total de multiplicación depende de la cantidad de ciclos necesarios para completar las operaciones de carga, desplazamiento y suma.

4.1 Caso 1(9×2)

Se realizó una prueba con la operación 9×2 para medir el tiempo de multiplicación. La multiplicación finaliza después de 168 ns, dado que, como se muestra en las *figuras 6 y 8*, es el momento en que la señal Done pasa de 0 a 1, indicando que el cálculo ha terminado. Como se observa en la figura 7, la multiplicación comienza tras 24 ns,

cuando Done cambia a 0 en un flanco de bajada del reloj, con STB en 1, lo que activa la máquina de estados. Por lo tanto, el tiempo total que lleva realizar la multiplicación es de 168 ns - 24 ns, resultando en 144 ns. Aunque Result podría contener el valor correcto antes de que Done valga 1, es esencial esperar este cambio para asegurar que el valor en Result sea confiable y libre de posibles interferencias o ruido.

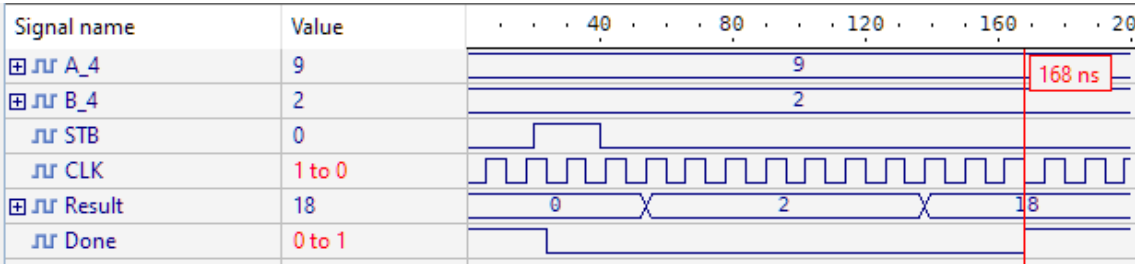


Figura 8: 9x2 en el instante antes de que done cambie de 0 a 1

4.2 Caso 2(2 × 9)

Para la operación inversa 2×9 , la multiplicación concluye después de 108 ns, tal como se muestra en las figuras 9 y 10, donde la señal Done cambia de 0 a 1, lo que indica que el cálculo ha finalizado. Sin embargo, como se observa en la figura 7, el proceso comienza 24 ns después del inicio, cuando Done pasa a 0 en un flanco de bajada del reloj y STB está en 1, activando así la máquina de estados. Por lo tanto, el tiempo total necesario para realizar la multiplicación es de 108 ns - 24 ns, resultando en 84 ns. Aunque el valor en Result podría estar presente antes de este cambio en Done, es necesario esperar a que Done sea 1 para asegurar que el valor en Result sea confiable y evitar posibles lecturas de datos no válidos.

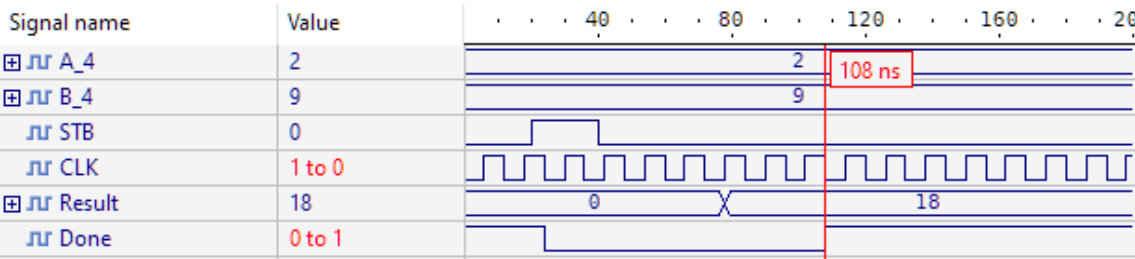
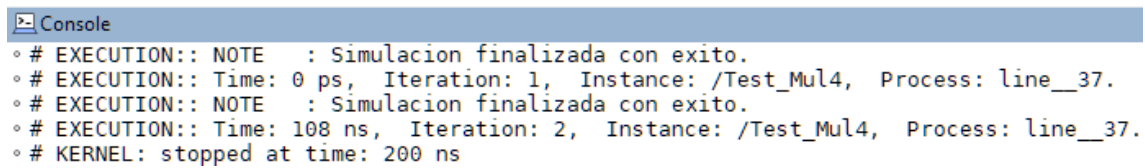


Figura 9: 2x9 en el instante antes de que done cambie de 0 a 1

A screenshot of a console window with a blue title bar labeled 'Console'. The window contains five lines of text: the first and third lines are '# EXECUTION:: NOTE : Simulacion finalizada con exito.'; the second line is '# EXECUTION:: Time: 0 ps, Iteration: 1, Instance: /Test_Mul4, Process: line__37.'; the fourth line is '# EXECUTION:: Time: 108 ns, Iteration: 2, Instance: /Test_Mul4, Process: line__37.'; and the fifth line is '# KERNEL: stopped at time: 200 ns'.

```
# EXECUTION:: NOTE : Simulacion finalizada con exito.
# EXECUTION:: Time: 0 ps, Iteration: 1, Instance: /Test_Mul4, Process: line__37.
# EXECUTION:: NOTE : Simulacion finalizada con exito.
# EXECUTION:: Time: 108 ns, Iteration: 2, Instance: /Test_Mul4, Process: line__37.
# KERNEL: stopped at time: 200 ns
```

Figura 10: Visualización del Note en la consola (2x9)

4.3 Cálculo de XX

Según la regla, dado que el número de legajo es mayor a 2000 y menor de 3000, se multiplica por 3 los dos primeros dígitos significativos (29). Esto da como resultado:

$$XX = 29 \times 3 = 87$$

5. Conclusión

Al comparar los dos casos, observamos que la multiplicación es 60 ns más rápida cuando el operando A es el menor (0010) en lugar de ser el mayor (1001). Esta diferencia en tiempo se debe a que un operando con menos bits significativos en 1 requiere menos desplazamientos y sumas parciales para completar la multiplicación, reduciendo así los ciclos de reloj necesarios. Este hallazgo destaca la importancia de asignar el operando con menos bits activos a A en aplicaciones donde se busca optimizar la velocidad de procesamiento.

En conclusión, al asignar el operando menor al valor de A, se logra una reducción en los tiempos de ejecución. En este caso, la diferencia se traduce en aproximadamente 5 ciclos de reloj, lo que representa una mejora relevante en rendimiento para aplicaciones que priorizan la eficiencia en operaciones aritméticas rápidas.