Diseño Jerárquico en VHDL

Un diseño complejo normalmente consta de varias descripciones más pequeñas llamadas bloques funcionales. Esto hace que los diseños sean más organizados, más fáciles de depurar, es decir de encontrar errores y más fáciles de simular. En un diseño esquemático el diseño jerárquico se hace usando los símbolos: se crea un circuito, se le hace el símbolo y luego se utiliza este símbolo en un circuito de mayor nivel. En VHDL el procedimiento es parecido pero en este lenguaje se le llama ***componente*** al circuito de menor jerarquía. En la Tabla 1 se muestra la comparación de términos entre el diseño esquemático y los términos usados en VHDL. En el caso de un diseño esquemático, después de creado un símbolo este se pone en un una nueva plantilla para su conexión, en el caso de un código VHDL esta misma acción se convierte en declarar el componente en el código y ***mapearlo*** en el mismo código para su conexión.

Término en diseño esquemático	**Equivalente en VHDL**
Símbolo	Componente
Construir el circuito del símbolo.	Escribir el código VHDL del componente.
Crear el símbolo.	Declarar el componente en el código
Pegar el símbolo en la plantilla	Mapear el componente en el código.

Tabla 1. Comparación de términos entre el diseño esquemático y los términos usados en VHDL.

Veamos un ejemplo. En la Figura 2 se muestran las operaciones que debe hacer una Unidad Aritmético-lógica (ALU) específica. Una ALU es un circuito que puede hacer varias funciones pero no al mismo tiempo, es decir el circuito tiene una sola salida y entonces debe existir un selector de funciones que escoja la función a desarrollar.

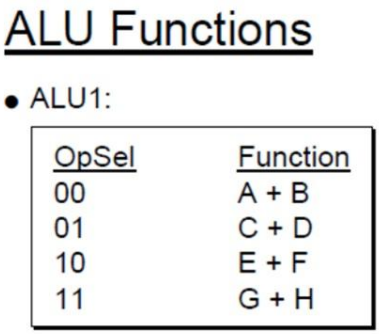


Figura 2. Funciones de una ALU específica. Cortesy of SynthWorks & ALDEC

En la Figura 3 se puede observar dos posibles soluciones de la ALU de la Figura 2. Se observa que ambas usan sumadores y multiplexores únicamente. En términos de símbolos piense un

momento ¿Cuántos símbolos se necesitan para el circuito 1 y cuántos símbolos se necesitan para el circuito 2? Tómese un momento para pensar y anote su respuesta antes de seguir leyendo.

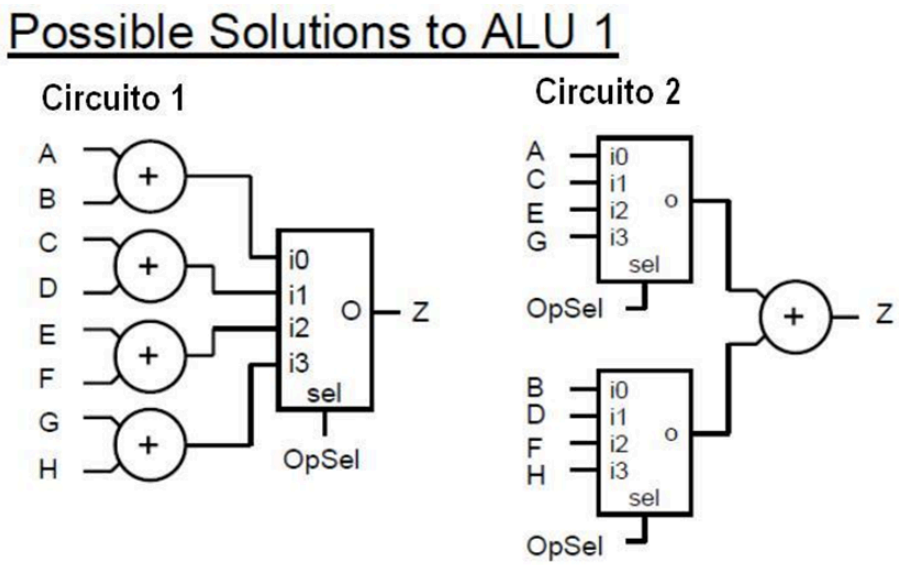


Figura 3. Dos posibles soluciones de la ALU en la Figura 2. Cortesy of SynthWorks & ALDEC

Si dijo que para el circuito 1 se necesitan 5 símbolos (4 de suma y un multiplexor) y que para el circuito 2 se necesitan 3 símbolos (dos multiplexores y un sumador) usted está totalmente errado. Ambos circuitos usan sólo dos símbolos. Si se implementará este circuito en el ISE Foundation de Xilinx se construirán dos circuitos: un solo multiplexor y un solo sumador. Ahora, si se quiere implementar el circuito 1, habría que usar el símbolo del sumador 4 veces y el símbolo del multiplexor una vez. Si se quiere implementar el circuito 2, habría que usar el símbolo del sumador una vez y el símbolo del multiplexor dos veces. Pero en ambos casos son solo dos símbolos usados diferentes veces. De la misma forma en código VHDL se hacen dos códigos uno para el sumador y otro para el multiplexor. En otro código de mayor jerarquía se declaran el sumador y el multiplexor como componentes (solo dos componentes) y se mapean las veces que se necesiten. Por ejemplo: para el circuito 1 el sumador se declara como componente una vez y se mapea 4 veces, el multiplexor se declara como componente una vez y se mapea una vez; en el circuito 2 el sumador se declara como componente una vez y se mapea una vez, y el multiplexor se declara como componente una vez y se mapea 2 veces.

Piense por un momento ¿Cuál es el circuito más óptimo para implementar la aplicación, el circuito 1 o el circuito 2? Un circuito en una FPGA es óptimo por área (El circuito es mucho mejor si es más pequeño) o por velocidad (El circuito es mucho mejor si tiene menos retardo).
Dado que los circuitos utilizan el mismo tipo de sumadores y multiplexores, piense por un momento qué circuito tiene menos retardo. Tómese el tiempo necesario para analizar y solo siga leyendo cuando termine su análisis y ya tenga una respuesta.

-
-
-

Si dijo que el el circuito 1 tiene más retraso porque tiene más componentes, usted está totalmente equivocado. Ambos circuitos tienen el mismo retardo = Retardo de un sumador + Retardo de un multiplexor. Recuerde que los circuitos trabajan en paralelo, entonces todos los multiplexores trabajan al mismo tiempo y todos los sumadores trabajan al mismo tiempo.

Ahora analice qué circuito tiene menos área, o que es lo mismo preguntarse qué circuito tiene menos compuertas. Tómese el tiempo necesario para analizar y solo siga leyendo cuando termine su análisis y ya tenga una respuesta.

Esta respuesta depende del tamaño de los datos de entrada. Supongamos que los datos de entrada y salida son de 8 bits, ya que un sumador de 8 bits es mucho más complicado que un multiplexor de 4 datos de 8 bits entonces se concluye que el circuito 2 es más óptimo. Entonces se mostrará el código VHDL del circuito 2 y el código del circuito 1 se deja como ejercicio. Primero, recordemos que los códigos VHDL del multiplexor y del sumador ya están escritos y se encuentran en las diapositivas Combinational building blocks in VHDL. Los códigos se copian como referencia en la Tabla 2.

```vhdl
entity mux is
generic (n: NATURAL :=8);
port (x1, x2, x3, x4:
        in std_logic_vector (n-1 downto 0);
      sel: in std_logic_vector(1 downto 0);
     f: out std_logic_vector (n-1 downto 0));
end entity mux;

architecture mux1 of mux is
begin
   Selection: process (sel, x1, x2, x3, x4)
    begin
        case sel is
                when "00" => f <= x1;
                when "01" => f <= x2;
                when "10" => f <= x3;
                when "11" => f <= x4;
     when others => f <= (others => 'X') ;
        end case;
end architecture mux1;
```

```vhdl
entity nBitAdder is
    generic (n: NATURAL :=4);
    Port (A,B: in std_logic_vector (n-1 downto 0);
                                    Sum: out
std_logic_vector (n-1 downto 0);
                    Cout: out std_logic);
end entity nBitAdder;

Architecture unsigned of nBitAdder is
    signal result : std_logic_vector (n downto 0);
Begin
  result <= ('0'&A) + ('0'&B); --add everything
  Cout<= result(n);        --generate carry out
  Sum<= result( n-1 downto 0); --output connection
end architecture unsigned;
```

Tabla 2. Códigos VHDL de un multiplexor de 4 entradas y de un sumador génerico.

La entidad de la ALU se copia en la Tabla 3. En la entidad se observan las 8 entradas de datos de 8 bits cada una, el selector de función *opsel* de dos bits y la salida *Z* de 8 bits

```vhdl
entity Optimal_ALU is
        port (A,B,C,D,E,F,G,H: in std_logic_vector (7 downto 0);
                opsel: in std_logic_vector (1 downto 0);
                  Z: out std_logic_vector (7 downto 0));
  end Optimal_ALU;
```

Tabla 3. Entidad de la ALU óptima.

La Tabla 4 muestra la arquitectura de la ALU óptima. Nótese que hay dos señales declaradas que como su nombre lo indican serán las salidas de los dos multiplixores. Se observa también la forma de declarar los componentes. Obsérvese que un componente se declara antes del ***Begin*** de la arquitectura, en el mismo lugar donde se declaran las señales. Concéntrese también en la forma de como declara un componente. La forma de declaración es muy similar a la declaración de la entidad que representa. Compare por ejemplo la declaración del componente ***mux*** en la Tabla 4. y la entidad ***mux*** en la Tabla 2. La únicas dos diferencias estan

en el inicio y en el final. En la entidad se inicia con la palabra reservada ***entity*** y en el componente se inicia con la palabra reservada ***component***. Mientras que la entidad finaliza con ***end entity,*** un compomente finaliza con ***end component***. El resto del código es igual en la entidad y en el componente. Se sugiere hacer un "copy & paste" de la entidad cuando se declare un componente y editar el inicio y el final, esto con el fin de no cometer errores en la declaración del componente. Observe las mismas diferencias entre la entidad del sumador en la Tabla 2. y la declaración del componente en la Tabla 4.

```
architecture netlist of Optimal_ALU is
--Signal declarative
   signal muxout1, muxout2: std_logic_vector (7 downto 0);

--Component declarative
component mux is
    generic (n: NATURAL :=8);
    port (x1, x2, x3, x4: std_logic_vector (n-1 downto 0);
                    s: in std_logic_vector(1 downto 0);
                     f: out std_logic_vector (n-1 downto 0));
   end component mux;  --continua en la siguiente página

component nBitAdder is
    generic (n: NATURAL :=4);
     Port (A,B: in std_logic_vector (n-1 downto 0);
              Cin: in std_logic;
          Sum: out std_logic_vector (n-1 downto 0); Cout: out std_logic);
    end component nBitAdder;
-- fin de la declaración de componentes

Begin
   First_Mux: mux  port map  (A,C,E,G,opsel,muxout1);
   Second_Mux: mux port map (B,D,F,H,opsel,muxout2);
   addition: nBitAdder generic map (8) port map (muxout1,muxout2,'0',Z,open);
-- se pudo haber escrito también Z <= muxout1+muxout2;
end netlist;
```

Tabla 4. Arquitectura de la ALU óptima

Después del ***Begin*** en la Tabla 4 se escribe el mapeado de los componentes. La estructura de este mapeado es la siguiente:

"Nombre del mapeado" : "Nombre del componente" ***port map*** (conexión de señales y/o puertos);

El nombre del mapeado es oblgatorio puesto que esto lo diferencia de otros mapeos del mismo componente. El nombre del componente junto con las palabras reservadas ***port map*** son obligatorias también. Entre los paréntesis se escriben las señales y/o puertos de la entidad que

se quieran conectar con el componente. En la Tabla 4 obsérvese que las señales que se escriben entre el paréntesis en el mapeado ***Fisrt_Mux*** son *A, C, E, G, opsel y muxout1* pertenecen a la entidad ***Optimal_ALU.*** Estas señales se conectan en el mismo orden de los puertos del componente con en el orden en el que aparecen en el mapeado. Así, por ejemplo:

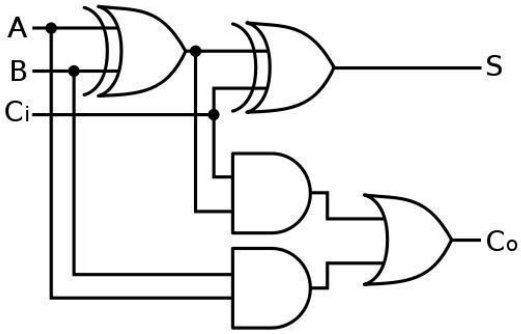
- **A** (puerto de la entidad ***Optimal_ALU***) se conecta con ***x1*** *(*puerto del componente ***mux***).
- ***C*** (puerto de la entidad ***Optimal_ALU***) se conecta con ***x2*** (puerto del componente ***mux***).
- ***E*** (puerto de la entidad ***Optimal_ALU***) se conecta con ***x3*** (puerto del componente ***mux***).
- ***G*** (puerto de la entidad ***Optimal_ALU***) se conecta con ***x4*** (puerto del componente ***mux***).
- ***opsel*** (puerto de la entidad ***Optimal_ALU***) se conecta con ***s*** (puerto del componente ***mux***).
- ***muxout1*** (señal de la entidad ***Optimal_ALU***) se conecta con ***f*** (puerto del componente ***mux***).

Observe en la Tabla 4. el mapeado de la suma (***addition***) tiene un mapeado del genérico (***generic map***) de 8. Esto quiere decir que cuando el software haga la síntesis o la simulación reemplazará el valor del genérico por el número 8. En el caso de los multiplexores no se escribió porque el genérico del multiplexor es 8, pero si se hubiese escrito funcionaría correctamente también. Se sugiere siempre hacer mapeo del genérico cuando la entidad del componente tenga uno o varios genéricos. En caso de varios genéricos se mapean en estricto orden de aparición, así como se mapean las señales de la entidad en el port map. También se pueden conectar constantes en un mapeado. En el mapeado ***addition*** se conecta un 0 lógico en el puerto Cin del componente, puesto que en este caso la suma no tiene acarreo de entrada. Si una salida de un componente no se va a usar, entonces puede mapearse un ***open*** en el puerto de esta salida, como lo muestra también el mapeado ***addition*** en la Tabla 4 para el puerto ***Cout***. Observe que en el comentario final en la Tabla 4. se pudo haber evitado el mapeo de la suma, puesto que la suma sin acarreo de entrada ni de salida se puede implementar muy fácilmente. Simplemente, en este caso se suman las dos señales y se le asiga a la salida Z, así: ***Z <= muxout1 + muxout2.***

Simulaciones en VHDL

La estructura jerárquica en VHDL también sirve para hacer simulaciones. En general un código HDL designado para simular se le llama *Testbench*. Estos testbench son códigos no sintetizables, es decir su función no es la implementación de un circuito si no entregar diferentes combinaciones de valores a las entradas para que el diseñador después de la simulación pueda verificar el comportamiento del circuito. La característica principal de un testbench es que su entidad este vacía, de aquí el hecho que los testbench no son sintetizables, no tiene entradas ni salidas. En la arquitectura de un testbench se declara el código que se quiere simular como un componente y se describe el comportamiento de las señales de entrada de ese componente.

Para ver una simulación con todos sus componentes se mostrará un ejemplo a continuación. En la Figura 4. se muestra el circuito de un sumador completo y su código VHDL.




```
entity fulladder is
    Port (A,B, Ci: in std_logic
          S, Co: out std_logic);
end entity fulladdder;

Architecture behavioural of fulladder is
    signal X : std_logic;
Begin
   X <= A xor B;  S <= X xor Ci;
 Co <= (X and Ci) or (A and B);
end architecture unsigned;
```

Figura 4. Circuito y código VHDL de un sumador completo de un bit.

Para simular este circuito hay que construir las formas de onda de las señales de entrada. Lo ideal para un circuito combinacional es generar todas las combinaciones de las entradas. En este caso es muy fácil generarlas puesto que solo son tres bits de entrada (A, B y Ci), lo que equivale a 8 posibles combinaciones. Las señales que habría que generar se muestran en la Figura 5. Note que hay 8 combinaciones diferentes para los tres bits en la Figura 5. Observe también los tiempos en los que las señales cambian, ya que esto será de gran importancia cuando se muestre el código VHDL para la generación de estas señales.

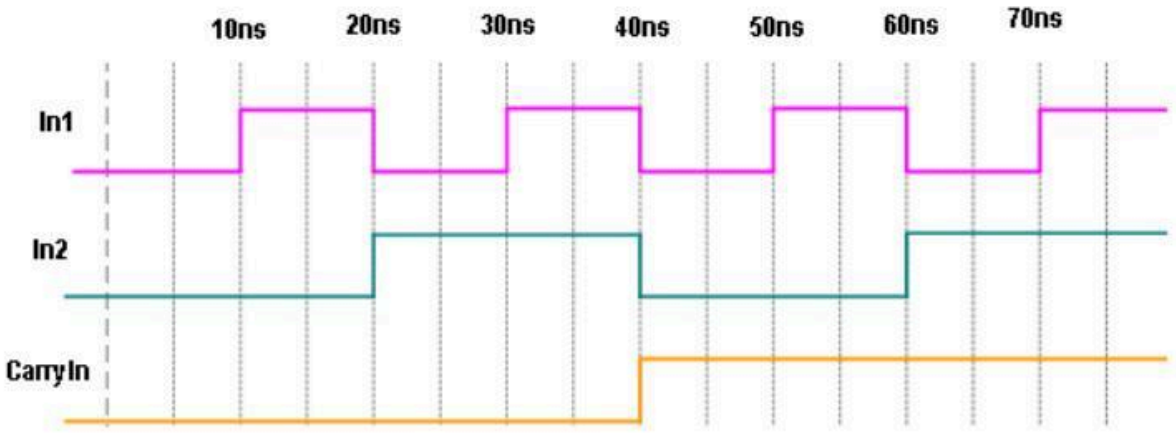


Figura 5. Waveform para simular un sumador completo de un bit.

En la Tabla 5. se muestra el código VHDL del testbench que genera las formas de onda de la Figura 5. y simula el sumador completo. Observe que la entidad esta vacía como se expresó anteriormente. El circuito a simular (sumador completo) se declara como componente y se declaran señales, una señal idéntica a cada entrada y salida del circuito, que servirán para poder generar las formas de onda requeridas. Notese que las señales que serán las entradas

del circuito son las únicas que tienen un valor inicial (**:= '0'**). Esto no es necesario pero es recomendable para que la simulación inicie con un valor por defecto

```
ENTITY Testbench1 IS  -- Note que la entidad esta vacía, no hay puertos de entrada ni de salida
END Testbench1;

ARCHITECTURE behavior OF Testbench1 IS

COMPONENT FullAdder
   PORT( A, B, Ci : IN std_logic;
         S, Co : OUT std_logic);
END COMPONENT;
--Inputs
    signal In1, In2, CarryIn : std_logic := '0'; --Inicialización de las entradas, en este caso en 0 para todas
--Outputs
    signal Total, CarryOut : std_logic;

BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: FullAdder PORT MAP (A => In1, B => In2, Cin => CarryIn,        S => Total, Co => CarryOut);

In1<= '1' after 10 ns, '0' after 20 ns, '1' after 30 ns, '0' after 40 ns, '1' after 50 ns, '0' after 60 ns, '1' after 70 ns;
In2<= '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
CarryIn<= '1' after 40 ns;
END;
```

Tabla 5. Testbench del sumador completo de la Fgura 5.

En la Tabla 5. observe como se mapea el componente a simular. Esta es otra forma de conectar las señales de una entidad con los puertos de un componente. A la izquierda se escribe el puerto del componente y seguido del símbolo => se escribe la señal del testbench que se quiere conectar con ese puerto del componente. Esta forma de mapeado tiene la ventaja que no es necesario guardar el orden de escritura de las señales del testbench, puesto que al ser expícita la conexión puerto => señal no se hace necesario el orden de aparición. Observe ahora la forma de hacer las formas de onda propiamente dicha. Se utiliza la palabra reservada **after** para indicar cuando se debe hacer la asignación que precede, así por ejemplo en el código **In2<= '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;** La señal In2 toma el valor de 1 lógico después de 20 ns, toma el valor de 0 después de 40 ns y toma el valor de 1 lógico después de 60 ns. Si se compara la forma de onda de In2 en la Figura 5. se observa que este código describe precisamente el comportamiento de la señal In2, y así también para las otras dos señales.

Después de ejecutarse la simulación el diseñador debe analizar los resultados y verificar que sean los esperados para el circuito en particular. En la Figura 6. se muestran los resultados de simular el sumador completo y puede verse que los resultados son los esperados.

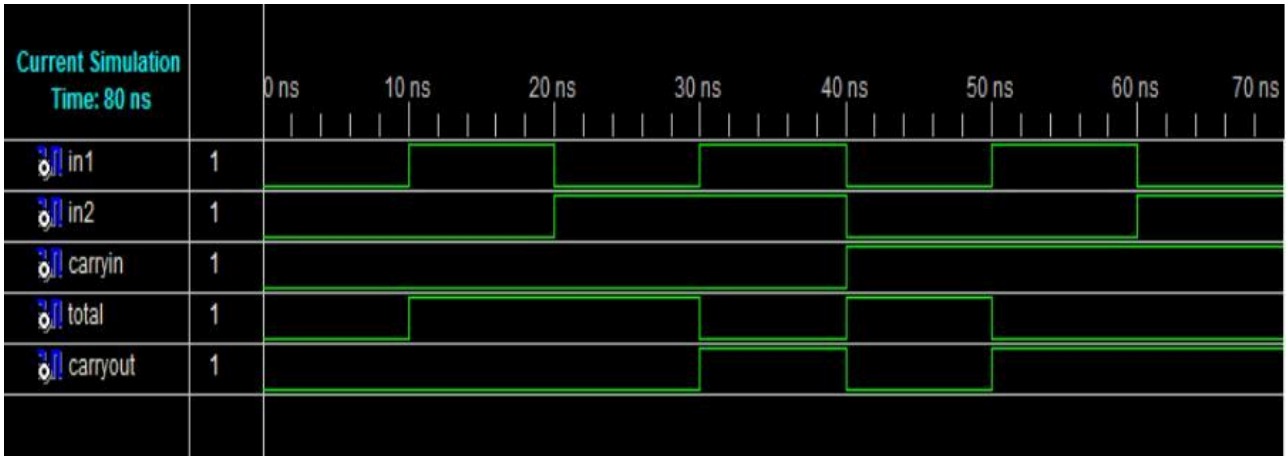


Figura 6. Resultados de la simulación del sumador completo

Algunas veces es muy díficil o engorroso simular todas las señales de un circuito combinacional. En estos casos es mejor intentar simular las funciones de los circuitos en algunos casos particulares de interes, poniendo bastante atención en las posibles singularidades que se pudieran presentar. Por ejemplo: Para simular un circuito multiplicador con signo de 8 bits cada dato se necesitarían 65536 combinaciones para simular todas las entradas posibles, lo que es muy tedioso con el método descrito anteriormente. Si A y B representan las entradas de este circuito, mejor sería escribir una simulación así A y B ambos positivos, A y B ambos negativos, A negativo y B positivo, A positivo y B negativo, además de las singularidades que puedan presentarse con el dato 0, tales como: A = 0 y B positivo y negativo, B = 0 y A positivo y negativo. Aunque esta simulación no asegura que todo el circuito este bien diseñado si puede entregar ideas de posibles fallas en el circuito. Otro ejemplo de un circuito donde no es necesario simular todas las combinaciones es el caso de un multiplexor. En este dispositivo es necesario simular todas las combinaciones del selector y pocos datos en las entradas de datos. En la Tabla 6. se muestra el testbench del multiplexor cuyo código aparece en la Tabla 2. Como se puede observar solo se simulan todas las combinaciones de *s* (dos veces), el cual es el selector. Para cada valor de *s* todas las entradas tienen valores diferentes para poder verificar que entrada se conecta con la salida. En la simulación se verifica que el valor de la salida sea igual al valor de la entrada que este direccionado por el selector *s* en ese momento. El resultado de la simulación se puede ver en la Figura 7. donde se puede apreciar el buen funcionamiento del multiplexor en todos los casos.

```
ENTITY Testbench1 IS
END Testbench1;

ARCHITECTURE behavior OF Testbench1 IS
  -- Component Declaration for the Unit Under Test (UUT)
component mux is
          generic (n: NATURAL :=8);
   port (x1, x2, x3, x4: in std_logic_vector (n-1downto 0);
                                    s: in std_logic_vector(1 downto 0);
                      f: out std_logic_vector (n-1 downto 0));
end component mux;

--Signal declaration
--inputs
    signal A,B,C,D : std_logic_vector (7 downto 0) := (others => '0');
    signal S : std_logic_vector (1 downto 0) := (others => '0');
--ouputs
    signal Y : std_logic_vector (7 downto 0);

BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: mux generic map (8) port map (x1=>A,  x2=>B, x3=>C, x4=>D, s =>S, f=>Y);
              A <= "00001010" after 5 ns, "10001010" after 40 ns;
          B <= "00001011" after 5 ns, "01001011" after 40 ns;
          C <= "00001100" after 5 ns, "00101100" after 40 ns;
          D <= "00001101" after 5 ns, "00011101" after 40 ns;
          S <= "01" after 10 ns, "10" after 20 ns, "11" after 30 ns,
            "00" after 40 ns, "01" after 50 ns, "10" after 60 ns, "11" after 70 ns;
END;
```

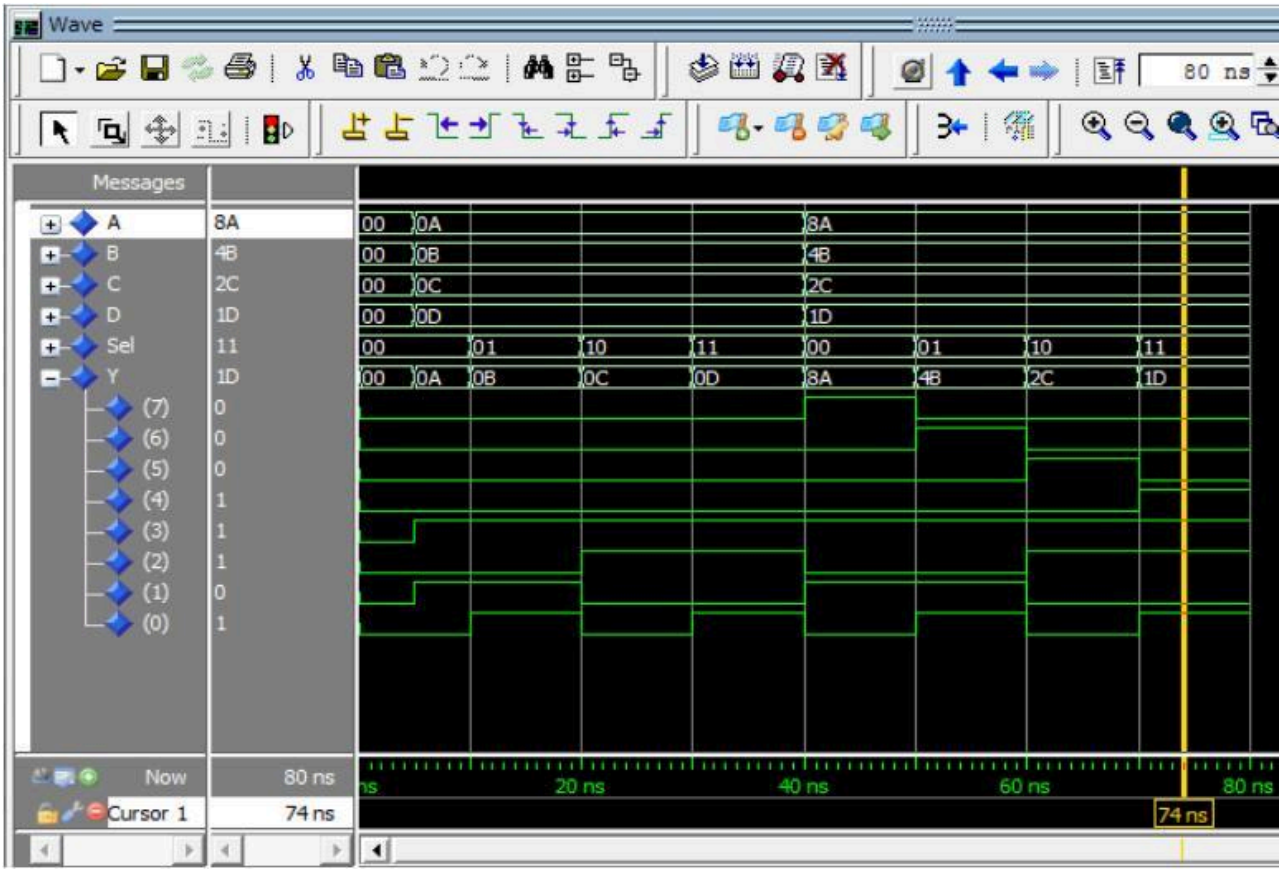


Figura 7. Simulación del multiplexor de 4 datos de 8 bits

Ejercicios de clase

1. Escriba la descripción del circuito 1 de la ALU de la Figura 3. usando como componentes el multiplexor y el sumador de la Tabla 2.
2. Escriba una simulación para la descripción del ejercicio 1. En la simulación debe asignar un dato diferente para cada entrada sin usar el dato 0 y debe simular todas las combinaciones del selector de función. Observe que de esta forma se simula toda la funcionalidad del circuito pero no se simulan todas las entradas.