25 de Agosto de 2020

Módulos Básicos

Laboratorio de arquitectura de Computadoras

ITESM Campus Monterrey

Profesor
Diego Fernando Valencia Martínez

Miguel Morales de la Vega
Javier Mondragón Martín del Campo

A00821541
A01365137

Introducción

Se realizaran 6 distintos módulos basicos para hacer la tarea final de realizar un CPU MIPS, el cual es una arquitectura computacional basada en RISC (Reduced Instruction Set Computer). Los 6 módulos a desarrollar son: Módulo de suma (Add), Módulo de desplazamiento lógico de dos bits para señales de 26 bits, Módulo de desplazamiento lógico de dos bits para señales de 32 bits, Módulo de “Program counter” (Flip flop tipo D con entrada de 32 bits), Módulo de incremento de tamaño de variable al doble (“Sign extender”) y módulos de multiplexores de 5 y 32 bits.

Procedimiento

Módulo 1. Add

Para la creación del módulo se hizo uso de las librerías básicas de operaciones aritméticas use *“IEEE.STD_LOGIC_1164.ALL;”*. La facilidad de este módulo se constituye dentro de las mismas instrucciones y la figura 1.1 ya que solo considera la operación de la suma y no toma en cuenta la definición de carry in y/o carry out.

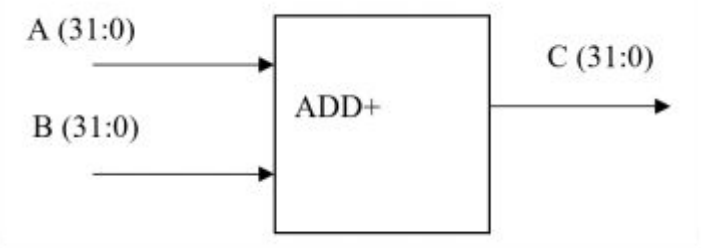

Figura 1.1. Módulo ADD para 32 bits.

La elaboración del código consta de un *“entity”* con sus respectivas entradas y salidas y una *“arquitecture”* simple y que no requiere el uso de un *“process”*, esto permite que el código con menor número de instrucciones y un mayor rendimiento dentro de su operación.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is
    Port ( NUM1 : in  STD_LOGIC_VECTOR (31 downto 0);
           NUM2 : in  STD_LOGIC_VECTOR (31 downto 0);
           SUM : out  STD_LOGIC_VECTOR (31 downto 0));
end adder;

architecture Behavioral of adder is
begin

    SUM <= NUM1 + NUM2;

end Behavioral;
```

Figura 1.2. Código de módulo ADD.

Módulo 2. Shift Left 2 bits para señales de 26 bits

El módulo obtiene de entrada una señal de 26 bits donde agrega en la salida dos ceros en el bit menos significativo, la señal resultante siempre será la señal de entrada mas dos ceros como se explicó anteriormente pero de igual forma el tamaño de la señal será de 28 bits (Figura 1.3).

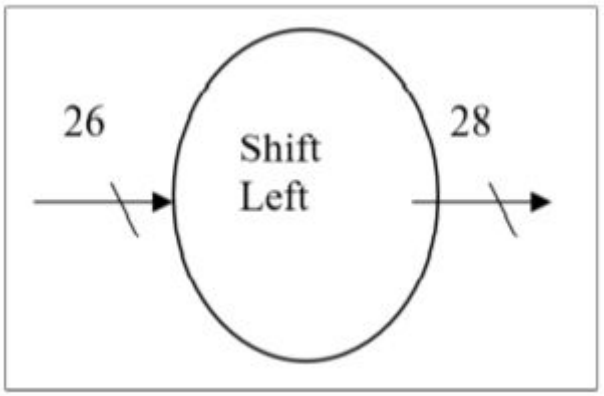

Figura 1.3
Diagrama del módulo 2

Para la implementación, se utilizó la operación de concatenación de el lenguaje VHDL representado con un “&”, anexando dos bits en 0, la implementación fue exitosa en un inicio y no se tuvieron que realizar cambios.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ShiftL2bits is
    Port ( D : in STD_LOGIC_VECTOR (25 downto 0);
           Q : out STD_LOGIC_VECTOR (27 downto 0));
end ShiftL2bits;

architecture Behavioral of ShiftL2bits is

begin

   Q <= D & "00";

end Behavioral;
```

Figura 1.4
Implementación del módulo 2

Módulo 3. Shift Left 2 bits para señales de 32 bits
El módulo usa operaciones lógicas para poder ejecutar el corrimiento hacia la izquierda con dos ceros. La primera instancia de creación de código se pensó en el uso de una librería y

la llamada de una función de corrimiento pero al ver que esta tomaba más tiempo de ejecución que una simple operación lógica se decidió cambiar.

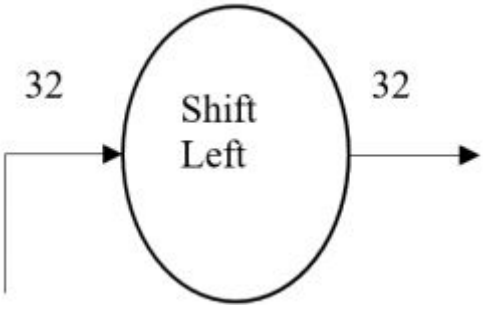

Figura 1.5. Módulo Shift Left para señales de 32 bits.

El módulo ejecuta una operación de concatenación con dos ceros y esta se pasa a la variable de salida. Los elementos que conforman el corrimiento son una entrada y una salida y la operación concatenación.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Shifter is
    Port ( VARIN : in  STD_LOGIC_VECTOR (31 DOWNTO 0);
           VAROUT : out  STD_LOGIC_VECTOR (31 DOWNTO 0));
end Shifter;

architecture Behavioral of Shifter is
begin

   VAROUT <= VARIN(29 downto 0) & "00";

end Behavioral;
```

Figura 1.6. Código de módulo Shift Left.

Módulo 4 Program counter

Este módulo es implementar un Flip Flop tipo D donde obtenga en la entrada 32 bits y al capturar un cambio decreciente en el reloj, esté guardará en la salida lo que haya en la entrada hasta obtener una señal de uno lógico en RESET, donde regresará los bits a un valor en 0 (Figura 1.7).

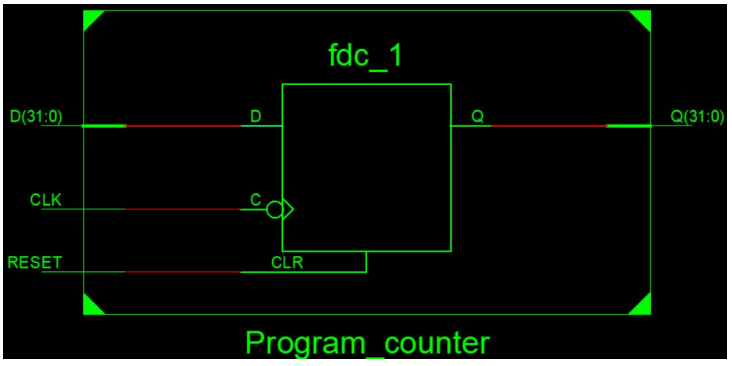

Figura 1.7
Esquemático del módulo 4

Para la implementacion del codigo se hizo uso de los procesos en VHDL implementando la función “if and else”, VHDL traducía el proceso al flip flop pero no era lo mas optimo, para

hacer un buen manejo del sistema se utilizo la funcion when que ya no requería de procesos secuenciales para el funcionamiento del módulo (Figura 1.8).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Program_counter is
    Port ( D : in  STD_LOGIC_VECTOR (31 downto 0);
           Q : out  STD_LOGIC_VECTOR (31 downto 0);
           RESET : in  STD_LOGIC;
           CLK : in  STD_LOGIC);
end Program_counter;

architecture Behavioral of Program_counter is

begin

   Q <= (others => '0') when RESET = '1' else
      D when falling_edge(CLK);

end Behavioral;
```

Figura 1.8
Implementación del módulo 4

Módulo 5. Sign extender

Para la creación del módulo Sign Extender se hizo una búsqueda de cómo poder extender la entrada de 16 bits a una salida de 32 respetando el signo que tiene la entrada. La ejecución del módulo es sencilla si el bit más significativo de la entrada es un 1 se extenderá con 1 hasta conformar 32 bits, si el bit más significativo es 0 se extenderá con ceros hasta conformar 32 bits.

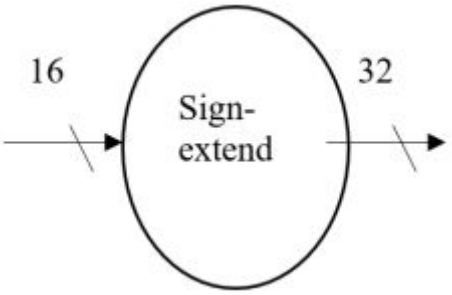

Figura 1.9 Módulo Sign Extender para señales de 16 bits.

En la sección del código que se muestra en la figura 1.10 se puede observar el uso de una función *“resize”* la cual está integrada dentro de la librería de *“NUMERIC_STD”*. La función tiene el siguiente esquema, *“function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;”* donde el valor *“ARG”* es nuestro dato de entrada y *“NEW SIZE”* seria el tamaño al cual queremos extender nuestra entrada, podemos observar que esta función es de tipo *“signed”* lo cual realiza el llenada de bits expandidos con el valor del signo en *“ARG”*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity extender is
    Port ( A : in std_logic_vector (15 downto 0) := "1111000000001111";
           B : out std_logic_vector (31 downto 0) := (others => '0'));
end extender;

architecture behavorial of extender is
begin

    B <= std_logic_vector(resize(signed(A), B'length));

end architecture;
```

Figura 1.10 Código de módulo Sign Extender para señales de 16 bits.

Módulo 6 Multiplexores de 5 y 32 bits

El desarrollo de este módulo consiste en desarrollar dos multiplexores que al ser sensible a una entrada desplegaba una señal u otra dependiendo de la variable (Figura 1.11). Se desarrollaron dos multiplexores, uno para dos entradas de 5 bits y otro para dos entradas de 32 bits.

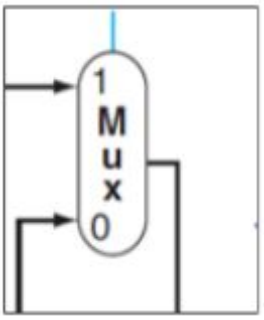

Figura 1.11
Esquemático del módulo 6

Al igual que el módulo 4, al principio se utilizó la lógica de un proceso y se implementaron “if and else” en la práctica pero por los mismos motivos se optó por la misma solución para terminar con una implementación muy sencilla mostrada en las figuras 1.12 y 1.13.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_5bits is
    Port ( D1 : in  STD_LOGIC_VECTOR (4 downto 0);
           D2 : in  STD_LOGIC_VECTOR (4 downto 0);
           S : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (4 downto 0));
end MUX_5bits;

architecture Behavioral of MUX_5bits is

begin

   Q <= D1 when S = '1' else D2;

end Behavioral;
```

Figura 1.12
Implementación de MUX de 5 bits

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_32bits is
    Port ( D1 : in  STD_LOGIC_VECTOR (31 downto 0);
           D2 : in  STD_LOGIC_VECTOR (31 downto 0);
           S : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (31 downto 0));
end MUX_32bits;

architecture Behavioral of MUX_32bits is

begin

   Q <= D1 when S = '1' else D2;

end Behavioral;
```

Figura 1.13
Implementación de MUX de 32 bits

Resultados

Módulo 1. Add

Para mostrar el correcto funcionamiento del módulo, se realizó un *"test bench"* con diferentes inputs al módulo. En la figura 2.1 se puede observar el resultado del *"test bench",* así como podemos ver los diferentes inputs al módulo, se observan salidas con el resultado correcto de la operación.

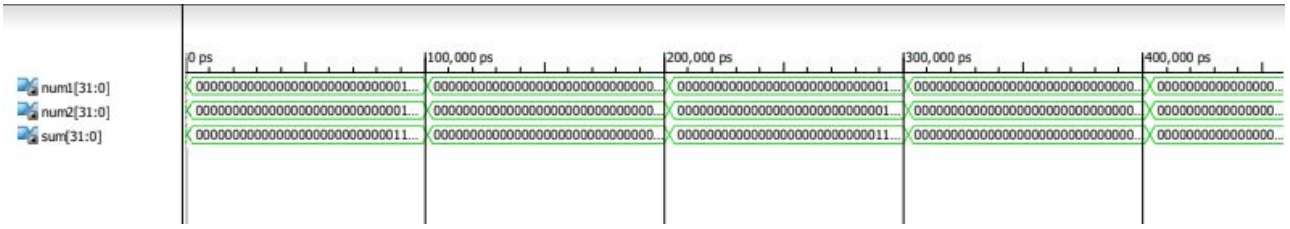

Figura 2.1. Test Bench módulo ADD para 32 bits.

Módulo 2. Shift Left 2 bits para señales de 26 bits

El funcionamiento correcto se puede apreciar en la figura 2.2, donde podremos observar que la salida es exactamente igual a la entrada con 2 bits a la izquierda en 0. Eso demuestra que está haciendo el corrimiento de bits de manera correcta.

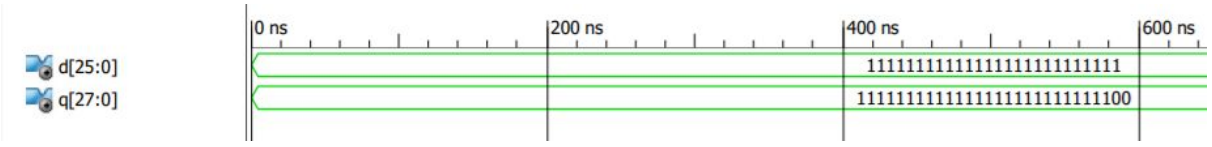

Figura 2.2
Resultados de testbench del módulo 2

Módulo 3. Shift Left 2 bits para señales de 32 bits

La ejecución correcta del módulo se encuentra en la figura 2.3, donde se puede observar que al valor de entrada se le agregan del lado derecho dos ceros realizando correctamente el corrimiento de los bits de entrada a la izquierda.

Figura 2.3. Test Bench módulo Shift Left para 32 bits.

Módulo 4 Program counter

Podemos observar que el funcionamiento es correcto en el circuito. Al principio se realiza un reset de los bits y la entrada se configura a un estado inicial. Al tener un flanco negativo la señal de la salida cambia a tener la de la entrada y esta no cambia hasta tener un uno lógico en la entrada “RESET” a regresar a 0. Este comportamiento se mantiene por todo el testbench.

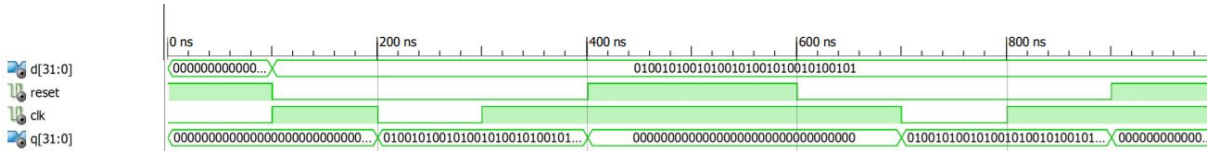

Figura 2.4

Resultados de testbench del módulo 4

Módulo 5. Sign extender

El funcionamiento correcto del módulo se puede apreciar con el *“Test Bench” mostrado en la figura 2.5, donde se puede apreciar como a una entrada con un signo de 1 es extendida y los bits expandidos son llenados con 1. También se puede observar el caso contrario donde los bit expandidos son llenados con 0 si la señal de entrada tiene un signo de 0.*

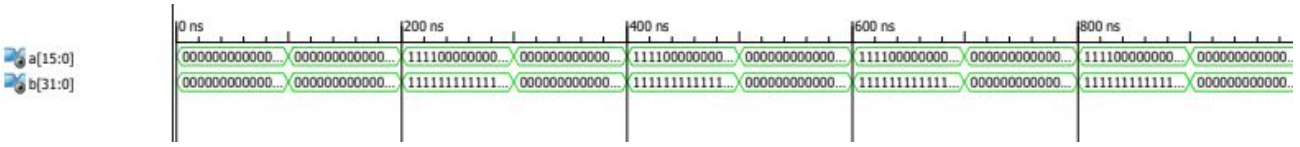

Figura 2.5. Test Bench módulo Sign Extend para señales de 16 bits..

Módulo 6 Multiplexores de 5 y 32 bits

Podemos observar en las figuras 2.5 y 2.6 que el funcionamiento es correcto en ambos multiplexores y al tener un uno lógico cambia la salida a la primera entrada y al tener un 0 lógico la salida es igual a la segunda entrada.

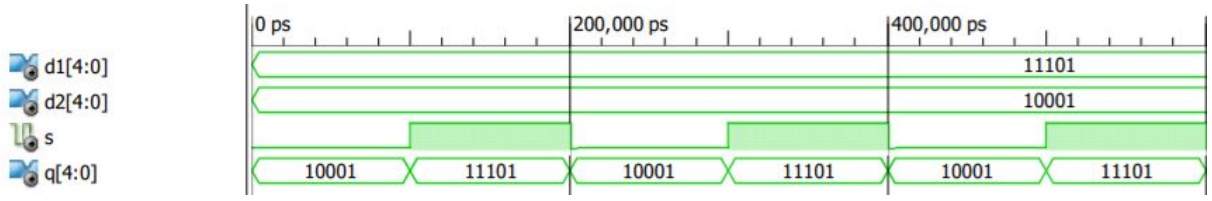

Figura 2.5

Resultados de testbench del módulo 6 para 5 bits

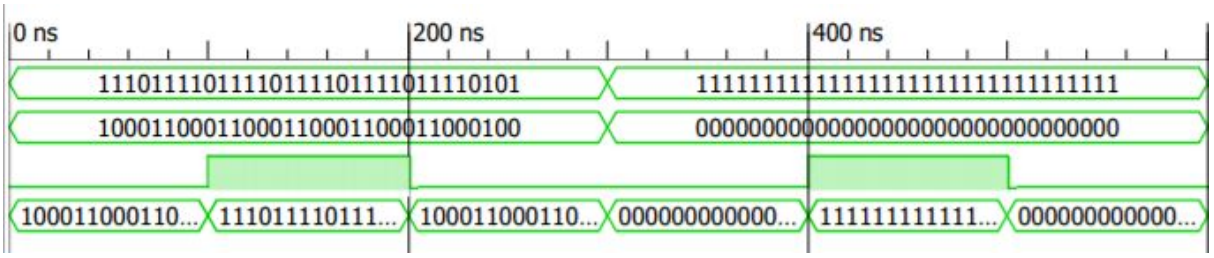

Figura 2.6
Resultados de testbench del módulo 2

Conclusión

Durante el desarrollo de la práctica pudimos visualizar los componentes básicos de las operaciones que realiza un procesador. La creación de estos módulos aun no nos pueden dar una imagen general de cómo se realizan las operaciones de una arquitectura computacional pero nos ayuda a entender que esta se conforma de pequeñas y simples operaciones.

Los módulos tienen que ser concurrentes, simples y que garanticen un gran desempeño, ya que al momento de unir el funcionamiento de los módulos es importante garantizar que no se presentarán errores y la arquitectura tendrá una complejidad no en el funcionamiento o el ensamblado de módulos si no en la construcción de nuevos componentes.

Reflexion Individual

Miguel Morales A00821541.
Durante el desarrollo de los módulos pude recopilar la forma en programar VHDL ya que en mi caso particular pasé más de dos años sin programar en este lenguaje. Estos módulos me dan cierta noción de las pequeñas operaciones que se necesitan en una arquitectura computacional pero aun no puedo visualizar cómo estos módulos podrían ensamblarse. De los aspectos más difíciles de la práctica fue el sign extend donde tuve que investigar más a fondo las funciones dentro de las librerías y ver cual podría apoyarme.

Javier Mondragon A01365137:
Durante el desarrollo de los módulos pudimos observar que VHDL genera los módulos de la mejor forma posible a pesar de su implementación, a pesar que no sea la manera más óptima. Al utilizar los módulos sin procesos cambia la mentalidad al programar y me hace regresar un año antes a cuando tomé la materia de sistemas digitales avanzados a como pensar mas con módulos e implementación de desarrollos asíncronos en lugar de la programación de un lenguaje de programación secuencial que es totalmente distinto y como reducir y simplificar instrucciones para utilizar menos librerías y hacerlo lo más simple posible.

Referencias

https://www.mips.com/products/architectures/
https://www.csee.umbc.edu/portal/help/VHDL/numeric_std.vhdl
https://stackoverflow.com/questions/17451492/how-to-convert-8-bits-to-16-bits-in-vhdl