UNIVERSIDAD NACIONAL DE TUCUMAN
FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA
DEPARTAMENTO DE ELECTRICIDAD, ELECTRONICA Y COMPUTACION

CÁTEDRA DISEÑO LÓGICO

Proyecto Integrador

Integrantes:

- Andrada, Luis Elian

Profesores:

- Mg. Ing. Mario de los Angeles Gómez Lopez
- Ing Carlos Sueldo
- Mg. Ing. Leonardo Daniel Del Sancio

Índice:

1)Introducción

1.1) Objetivo del proyecto

Este proyecto tiene como objetivo desarrollar un microprocesador con modificaciones específicas para facilitar su comprensión a personas sin conocimientos previos en el tema. Es especialmente adecuado para estudiantes que están en la transición entre los conceptos básicos de electrónica digital y la arquitectura de microprocesadores. Para lograr esto, hemos diseñado el microprocesador para ser controlado desde una computadora, utilizando la comunicación UART para enviar las instrucciones al microprocesador. A su vez, el resultado de las instrucciones se mostrará en una pantalla LCD, lo que lo hace más transparente y comprensible para el usuario.

1.2) Alcance del proyecto

Creemos firmemente que este tipo de proyectos contribuirá en gran medida a nuestra facultad y será especialmente beneficioso para los estudiantes que estén pasando de materias como Electrónica 2 a Electrónica 4. Aunque inicialmente teníamos la intención de desarrollar un procesador más complejo con un conjunto de instrucciones más amplio, decidimos incorporar funcionalidades como la pantalla LCD y la comunicación UART, lo que impidió la inclusión de características adicionales. Sin embargo, creemos que estas adiciones enriquecen la experiencia de aprendizaje al permitir una interacción más intuitiva con el microprocesador.

1.3) Metodologia usada

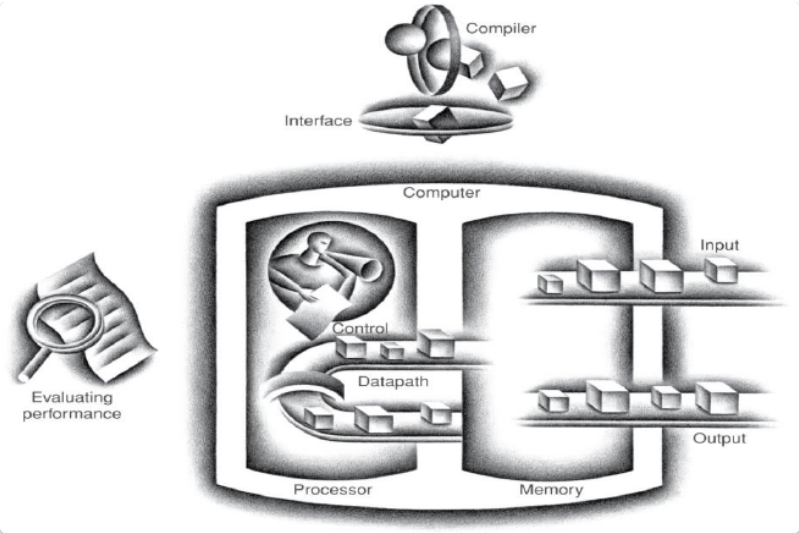
El método que utilicé para el diseño y verificación de estos bloques se llama "desarrollo basado en pruebas" (Test-Driven Development, TDD - ANEXO A). Al utilizar esta metodología, no es necesario verificar señal por señal para asegurarnos de que funcione correctamente, solo necesitamos verificar que no se produzcan errores. Es por eso que este informe no tendra seccion de simulacion y pruebas ya que solo me hizo falta correr el TB y verificar no tener ningun aviso

2) Base teórica

2.1) Partes de una computadora

Todas las computadoras tienen los mismos componentes básicos:

1. Procesador
   - Control: Maneja y sincroniza el camino de datos.
   - Camino de datos: Por donde se mueven los datos.
2. Memoria
   - Se comunica con el procesador (intercambio constante entre ambos).
   - Se comunica con el mundo exterior.
3. Sistema de entrada y salida.



2.1.1) Procesador

El procesador manipula los datos de la memoria. La CPU es básicamente una máquina de estado finito que sigue la secuencia "Leer -> Decodificar -> Ejecutar". Durante la etapa de ejecución, la CPU puede leer y escribir datos de memoria, recibir y enviar datos a dispositivos de entrada y salida, y realizar operaciones lógicas y aritméticas.

El procesador está compuesto por:

- Unidad de control: Máquina de estado finito que interpreta las instrucciones y genera señales de control.
- Camino de datos: Incluye registros, ALU y las interconexiones.

Unidad de control

La unidad de control es una parte esencial de la CPU que coordina y controla las operaciones internas. Actúa como el "cerebro" del procesador, interpretando instrucciones y generando señales de control para ejecutarlas. Interactúa con el camino de datos, encargado de las operaciones y el movimiento de datos. La unidad de control envía señales al camino de datos para indicar qué operaciones realizar y en qué orden. En resumen, la unidad de control interpreta y ejecuta instrucciones, mientras que el camino de datos realiza las operaciones necesarias. Ambos trabajan juntos para procesar instrucciones y datos en una CPU.

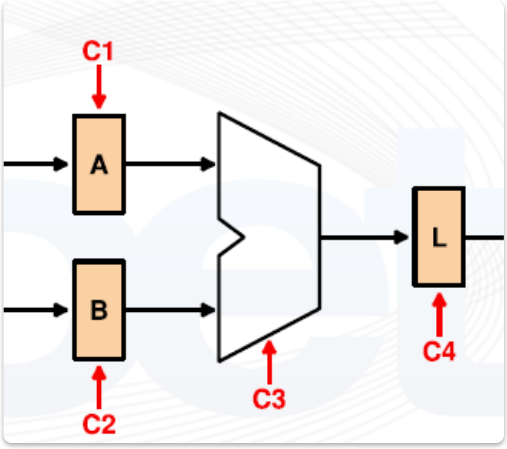
La unidad de control puede ser una máquina de estado finito o un simple bloque combinacional, dependiendo del camino de datos diseñado.

Componentes del camino de datos

ALU

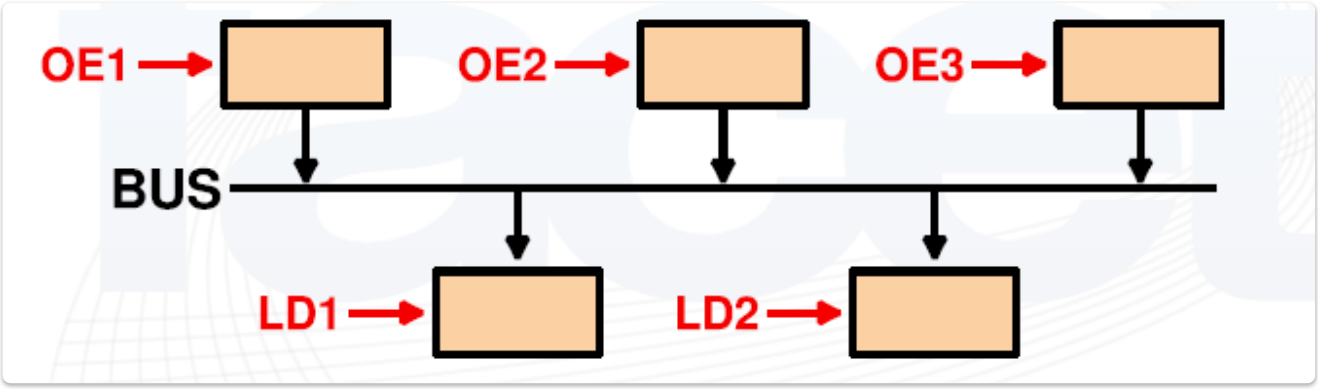
Una ALU (Unidad Lógica Aritmética, por sus siglas en inglés) es un componente fundamental en un microprocesador, encargado de realizar operaciones aritméticas y lógicas en los datos que se procesan. La ALU es la unidad que ejecuta las operaciones fundamentales, como sumar, restar, multiplicar y dividir, así como operaciones lógicas como AND, OR, NOT, entre otras. Estas operaciones dependen de las señales de control proporcionadas.

En el diagrama a continuación se pueden observar dos registros que almacenan valores que ingresan a la ALU, y el resultado se guarda en el registro L.



BUS

Un BUS, por su parte, es un sistema de interconexión utilizado en los microprocesadores para transferir datos entre diferentes componentes, como la memoria, la ALU y otros dispositivos de entrada y salida. El BUS es un conjunto de cables y líneas de control que permiten enviar y recibir datos en diferentes direcciones dentro del microprocesador.



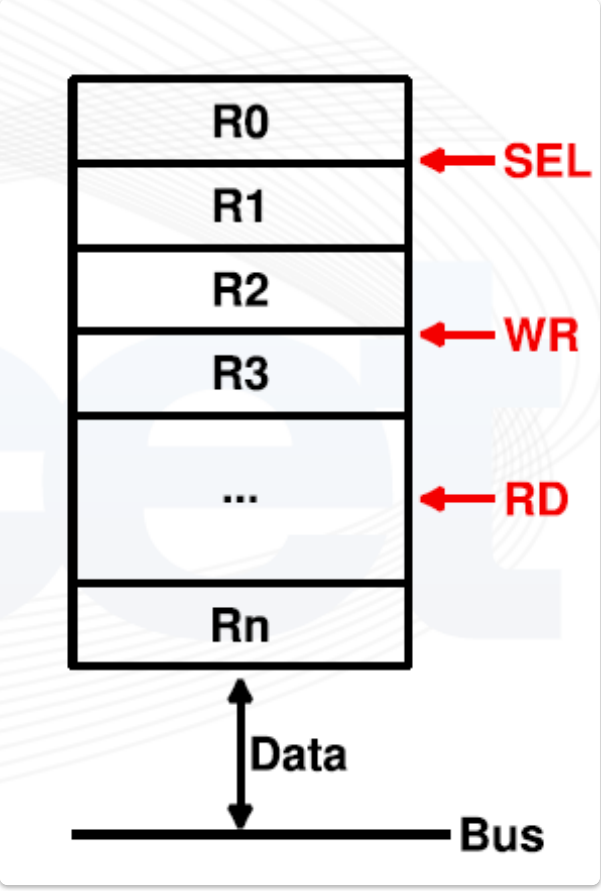
La grafica muestra como se puede subir y bajar datos de un bus. En este caso cuando algunas de las entrada de OE=1 se sube datos al bus y cuando LD=1 se baja datos del bus a los respectivos registros.

Banco de registros

Un banco de registros es un conjunto de registros de alta velocidad que se utilizan para almacenar temporalmente datos y resultados intermedios en un microprocesador. Los registros son un tipo de memoria interna de alta velocidad que se utilizan para realizar operaciones aritméticas y lógicas en los datos que se procesan.

El banco de registros se utiliza para almacenar los operandos de las operaciones que realiza la ALU, así como los resultados intermedios y finales. Los registros se organizan en grupos o bancos, y cada uno de ellos puede contener varios registros. El número de registros en un banco puede variar según el diseño del microprocesador.

El acceso a los registros es muy rápido, ya que se encuentran ubicados en el chip del microprocesador, lo que permite un acceso casi instantáneo a los datos almacenados en ellos. Además, el banco de registros permite que el microprocesador realice varias operaciones simultáneas, lo que puede mejorar significativamente su rendimiento.

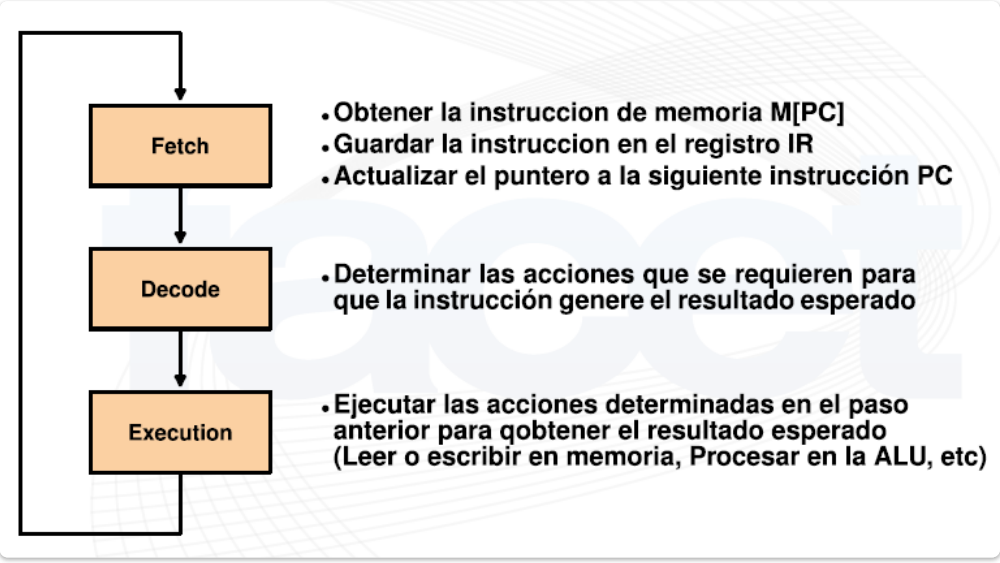


- Las entrada SEL(elije el registro), WR(Escribir) y RD(Leer) son las entradas de control
  - SEL requiere $log2(N)$ cantidad de lineas

2.2) Instrucciones

Las instrucciones en un CPU son comandos que indican al procesador qué operación realizar. Cada instrucción consta de un código de operación y posiblemente operandos. El CPU ejecuta las instrucciones secuencialmente, realizando operaciones como aritmética, lógica y transferencia de datos.

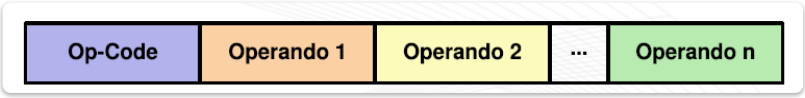
2.2.1)Ciclo de instrucciones



2.2.2) ¿Qué debe contener cada instrucción?

1. Qué operación realizar (Op-Code).

2. Dónde se guarda el resultado (Operando 1).
3. Dónde se encuentran los operandos (Operandos 2 en adelante).
4. Dónde está la próxima instrucción (siguiente línea de código).



2.2.3) Clases de instrucciones

- Movimientos de datos: Siempre tienen una fuente y un destino.
  - Load: Carga de datos desde memoria.
  - Store: Guarda datos en memoria.
- Procesamiento: Operan varios operandos y guardan el resultado en un destino.
  - add, sub, shift, etc.
- Control de flujo de instrucciones: Saltos que alteran el flujo de control normal.
  - Pueden ser condicionales o incondicionales.
- Misceláneas: Otras funciones que no se clasifican en los tipos anteriores.

2.2.4) Descripción formal de una instrucción (RTN)

RTN: Notación de transferencia de registros
La "Register Transfer Notation" (RTN) es una notación utilizada para describir el comportamiento de los circuitos digitales y los microprocesadores. En términos generales, la RTN describe cómo los datos se transfieren entre los registros de un microprocesador y los circuitos que los procesan.

En la RTN, se utiliza un conjunto de símbolos para representar las operaciones que se realizan en el circuito digital. Por ejemplo, un símbolo de flecha hacia la derecha indica una transferencia de datos de un registro a otro, mientras que un símbolo de suma representa una operación de suma.

Al utilizar la RTN para describir el comportamiento de un microprocesador, es posible crear un modelo abstracto que muestra cómo los datos se mueven a través del sistema. Esto es útil para diseñar y depurar circuitos digitales, ya que permite identificar posibles problemas y optimizar el rendimiento del sistema.

RTN abstracto: Describe qué hace la instrucción, pero no cómo se implementa.
RTN concreto: Describe cómo se implementa.

2.3) Modos de direccionamiento

Los modos de direccionamiento en un microprocesador se refieren a la forma en que la CPU accede a los datos o instrucciones almacenados en la memoria o en los registros internos del microprocesador. Estos modos varían según el tipo de ISA.

Existen varios tipos de modos de direccionamiento en los microprocesadores, entre ellos:

1. Modo de direccionamiento implícito: este modo se utiliza cuando la dirección de memoria está implícita en la operación que se está ejecutando. Por ejemplo, la instrucción "CLC" (limpiar el flag de acarreo) no requiere una dirección de memoria, ya que la operación se aplica directamente al registro de flags de la CPU.
2. Modo de direccionamiento inmediato: este modo se utiliza cuando los datos se encuentran en la misma instrucción que se está ejecutando. Por ejemplo, la instrucción "MOV AX, 1234H" mueve el valor inmediato 1234H al registro AX.

3. Modo de direccionamiento de registro: en este modo, la dirección de memoria se especifica utilizando el contenido de uno de los registros de la CPU. Por ejemplo, la instrucción "MOV AX, BX" mueve el contenido del registro BX al registro AX.
4. Modo de direccionamiento de registro indirecto: en este modo, se utiliza el contenido de un registro como dirección de memoria. Por ejemplo, la instrucción "MOV AX, [BX]" mueve el valor almacenado en la dirección de memoria contenida en el registro BX al registro AX.
5. Modo de direccionamiento indexado: en este modo, se utiliza un registro como índice para acceder a una dirección de memoria. Por ejemplo, la instrucción "MOV AX, [SI+10H]" mueve el contenido de la dirección de memoria resultante de sumar el contenido del registro SI más el valor 10H al registro AX.
6. Modo de direccionamiento rotativo: en este modo, se utiliza un registro como contador para acceder a una dirección de memoria que va rotando. Por ejemplo, la instrucción "MOV AX, [BX+SI+10H"] mueve el contenido de la dirección de memoria resultante de sumar el contenido de los registros BX más SI rotado al registro AX.

2.4) ISA

Las instrucciones de un microprocesador constituyen el lenguaje mediante el cual el programador le indica qué hacer. Además, definen las capacidades del procesador en un ciclo de instrucción, por lo que son fundamentales en el diseño del hardware.

2.4.1) ¿Qué es el ISA?

El ISA (Arquitectura de Conjunto de Instrucciones) es conocido como "el Modelo del Programador de la máquina". Es un nivel de abstracción que separa el desarrollo del software del desarrollo del hardware. Proporciona la información necesaria para escribir programas para un procesador específico.

2.4.2) Componentes del ISA

El ISA se compone de 4 componentes:

1. Celdas de almacenamiento

Todo ISA debe especificar sus celdas de almacenamiento, como el número y el tamaño de los registros de propósito general y especial del CPU, las celdas de propósito general en memoria y el almacenamiento relacionado con los dispositivos de E/S.

2. Formatos de las instrucciones

Los formatos de las instrucciones indican el tamaño y el significado de cada campo diferente en las instrucciones. Esto incluye los modos de direccionamiento soportados.

3. El conjunto (SET) de instrucciones

Es el conjunto completo de operaciones que puede realizar la máquina. Utiliza las celdas de almacenamiento, los formatos y los resultados del ciclo de instrucción.

4. Naturaleza del ciclo de instrucción

Esto incluye detalles finos sobre lo que se hace independientemente de la instrucción en cuestión.

2.4.3) Clasificación de ISAs

Los ISAs se clasifican según cómo y dónde se ubican los operandos y cómo se especifican estos operandos en la instrucción.

- Algunos registros tienen una "personalidad" propia, como el PC, los acumuladores y el puntero de la pila.
- Hay clasificaciones basadas en las instrucciones aritméticas que tienen dos operandos y un resultado.
- Los ISAs de más de 3 operandos no se ven actualmente.
  - ISA de 3 operandos: Utiliza modos de direccionamiento tanto para los operandos fuente como para el resultado.
  - ISA de 2 operandos: Utiliza un operando destino como fuente.
  - ISA de 1 operando: Emplea implícitamente un registro llamado acumulador.
  - ISA de 0 operandos: Comúnmente se maneja con los datos que están en la pila.

2.4.4) Falacias comunes sobre ISAs

- Instrucciones más poderosas no equivalen a mayor rendimiento. Pueden hacer que la ejecución sea más lenta debido a la dificultad del pipeline.
- Usar assembler no mejora el rendimiento. Los compiladores modernos son bastante buenos para los procesadores actuales.

Entonces, ¿para qué usamos assembler?

- Para comprender el funcionamiento del procesador.
- Para diseñar un buen compilador.
- A veces permite aprovechar al máximo el microprocesador sin necesidad de programar en él.

2.5) Pasos para diseñar una instrucción

1. Analizar el ISA.
2. Describir la instrucción en RTL abstracto.
3. Analizar el camino de datos para verificar si cumple con las especificaciones.
4. Modificar el control si es necesario.

2.6) Tipos de CPU

2.6.1) CPU multiciclo

Este tipo de CPU se divide en varias etapas y suele aprovechar esta división para aplicar una técnica llamada pipeline. Al dividirse de esta manera, tiene varias ventajas:

- Permite realizar aproximadamente 1 instrucción por ciclo.
- Su control es un circuito combinacional puro.

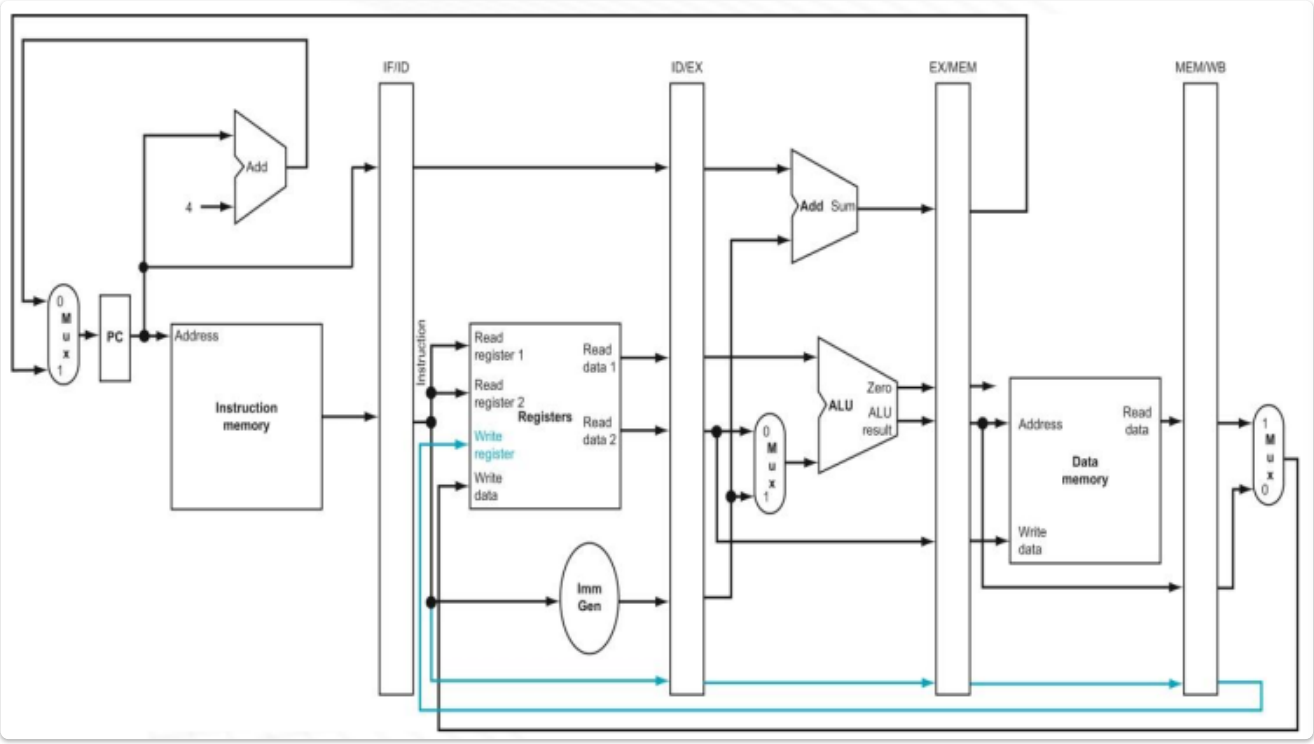
Sin embargo, también tiene varias desventajas que es importante destacar:

- Conflictos de datos (RAW).
- Conflictos con los saltos.

Hoy en día, todos los procesadores de alta performance son de este tipo. Además, agregan otras técnicas que esta presentación no nos permite explicar en detalle, como:

- Paralelismo a nivel de pipeline.
- Núcleos en paralelo.
- Ejecución especulativa, entre otras.

Ejemplo:



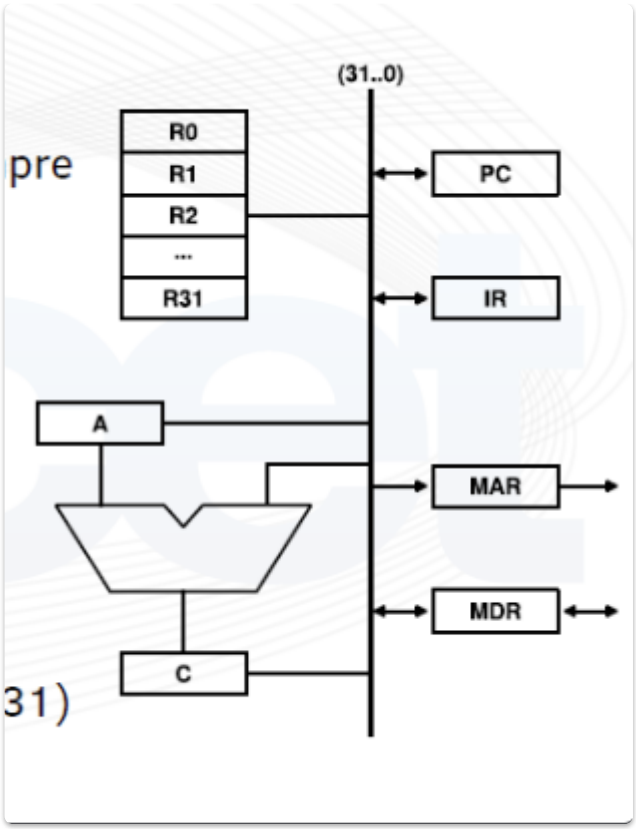
2.6.2) CPU multiciclo

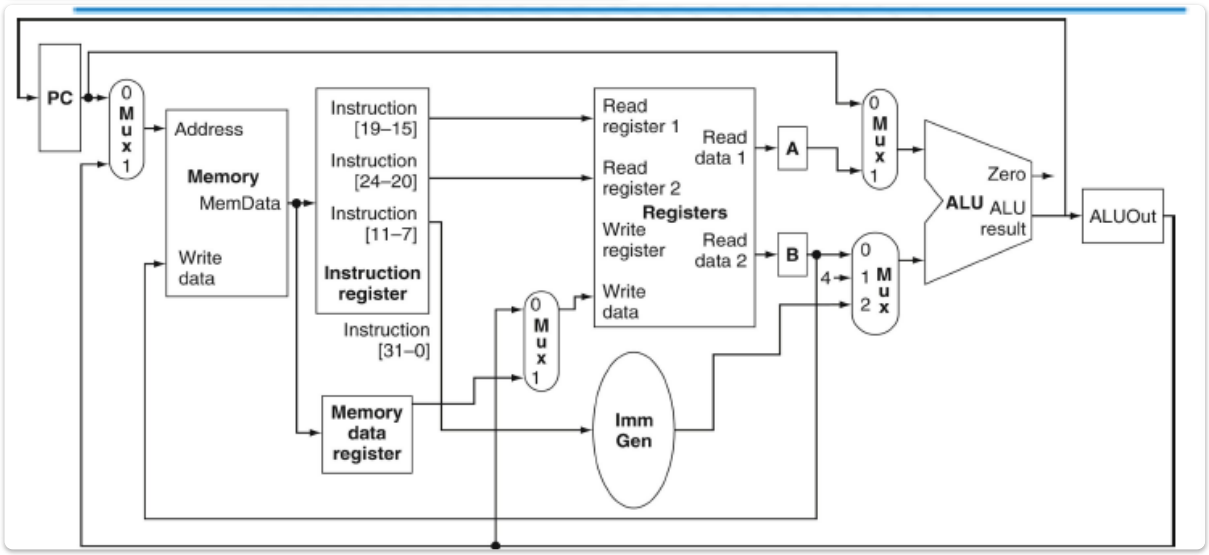
La idea básica del CPU multiciclo es dividir las instrucciones en múltiples "pasos" modulares. Cada instrucción se forma mediante la combinación de estos pasos, lo que significa que cada instrucción puede tardar una cantidad diferente de ciclos.

Los objetivos principales del diseño de un CPU multiciclo son:

- Ejecutar cada paso en un ciclo T pequeño.
- Lograr un balance entre los pasos para evitar cuellos de botella.
- Reutilizar los recursos para minimizar los costos.

Este tipo de CPU se utiliza en microprocesadores más pequeños y económicos.
Ejemplo:





3) Diseño teórico de nuestro proyecto

El objetivo de nuestro proyecto es crear un procesador multiciclo que pueda ser controlado desde una computadora mediante un puerto UART y que muestre el resultado en un LCD.

Desde la computadora, enviaremos una instrucción al CPU para indicar qué operación queremos que realice. Dado que el puerto UART utilizado en el curso solo puede enviar datos de 8 bits, dividiremos la instrucción en 2 partes. Luego, un bloque de entrada decodificará esta instrucción y enviará el código de operación a la unidad de control, la cual se encargará de manejar todo el camino de datos.

Además, para mejorar la experiencia del proyecto, agregamos un display de 7 segmentos y ajustamos las transiciones de la unidad de control para que ocurran cada segundo. Esto nos permitirá apreciar los cambios de estado y cómo funciona cada componente.

3.1) Definimos las componentes del camino de datos

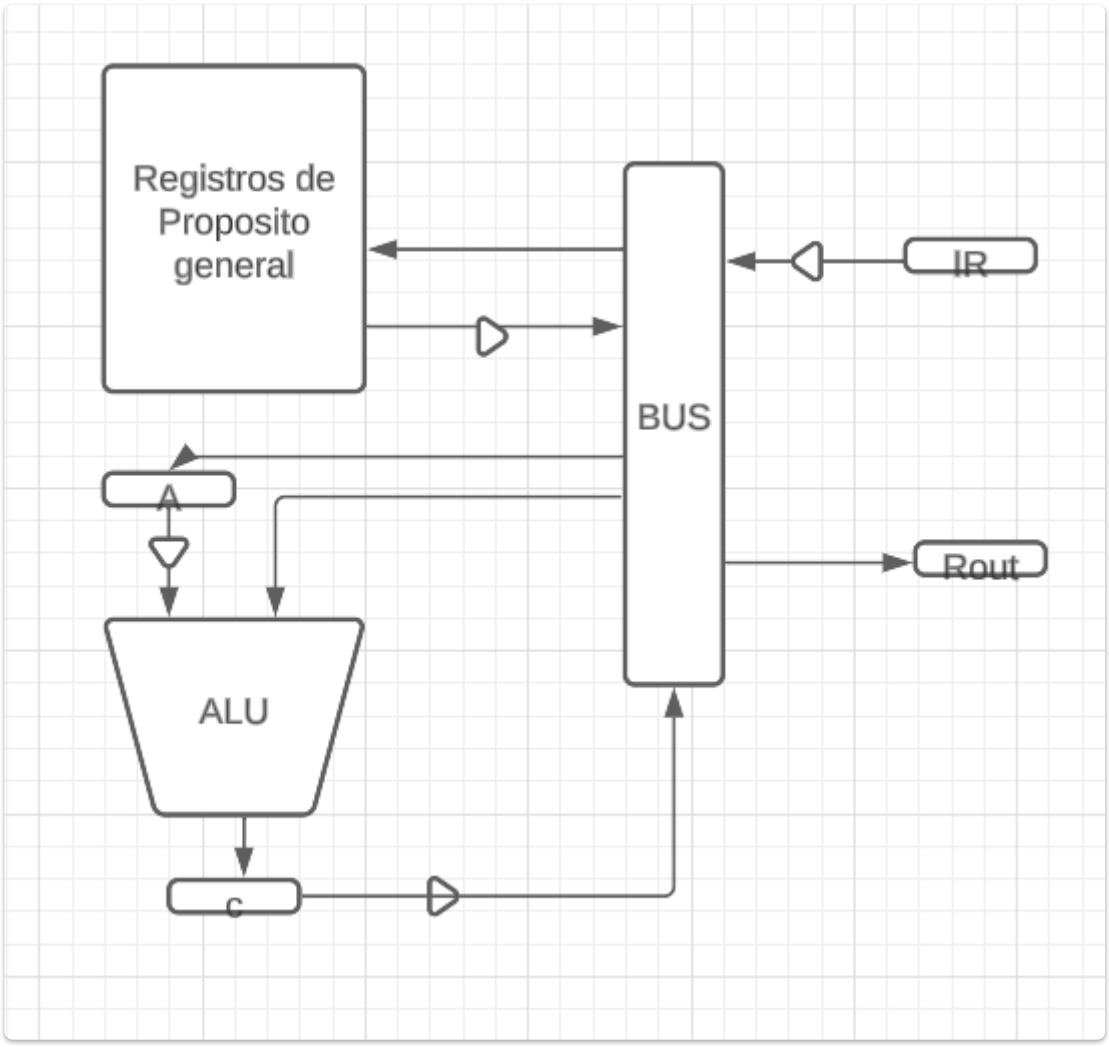
Intentaremos diseñar un CPU con la menor cantidad de recursos posibles, por lo que tendremos:

- 1 Banco de registros de propósito general con 8 registros.
- 1 ALU (Unidad Lógica Aritmética).
- 1 Registro para la entrada de la ALU.
- 1 Registro para la salida de la ALU.
- 1 BUS (sistema de interconexión).
- 1 Registro de salida.
- 1 Registro de entrada (IR).

3.1.1) Observaciones importantes:

- Todos los componentes que puedan enviar datos al BUS deben tener una salida con un buffer tristate para evitar conflictos.
- No necesitaremos memoria ya que la entrada UART funcionará como memoria de instrucciones y el registro de salida para el LCD funcionará como memoria de datos.
- Es importante tener en cuenta que este proyecto tiene fines didácticos y no busca implementar un CPU completo. En un diseño más completo, se requeriría la incorporación de memorias con sus respectivas jerarquías.

3.1.2) Conexión de nuestro camino de datos



3.2) Definimos el ISA que utilizaremos

3.2.1) Celdas de almacenamiento

- Tendremos registros de propósito general de 32 bits cada uno.
- Adicionalmente, un registro de entrada donde estará la instrucción actual (IR).
- Un registro de salida para mostrar el valor de algún registro.

3.2.2) Formato de las instrucciones

Nuestras instrucciones tendrán el siguiente formato:

Constante	Registro r1	Registro rs	OpCode

- Constante: 6 bits
- Registro r1: 3 bits para ubicarlo en el banco de registros.
- Registro rs: 3 bits para ubicarlo en el banco de registros.
- OpCode: 4 bits.

3.2.3) Conjunto de instrucciones

Tendremos un total de 10 instrucciones:

- LOAD: Cargar el valor de la constante en el registro rs.
- STORE: Cargar el valor del registro rs en el registro de salida.
- AND registro-registro: Realizar la operación AND entre el registro rs y r1.
- OR registro-registro: Realizar la operación OR entre el registro rs y r1.
- ADD registro-registro: Realizar la operación ADD entre el registro rs y r1.
- SUB registro-registro: Realizar la operación SUB entre el registro rs y r1.
- AND registro-constante: Realizar la operación AND entre el registro rs y la constante.
- OR registro-constante: Realizar la operación OR entre el registro rs y la constante.
- ADD registro-constante: Realizar la operación ADD entre el registro rs y la constante.
- SUB registro-constante: Realizar la operación SUB entre el registro rs y la constante.

3.3) Describimos el RTL abstracto de las instrucciones

Instrucción	RTL abstracto
LOAD	R[rs] <- Constante
STORE	Rout <- R[rs]
Aritméticas registros	R[rs] <- R[rs] operación R[r1]
Aritméticas registro-constante	R[rs] <- R[rs] operación Constante

3.4) Analizamos el camino de datos con el RTL concreto

LOAD

1. $BUS \leftarrow R_{in} : R[rs] \leftarrow BUS$

STORE

1. $BUS \leftarrow R[rs] : R_{out} \leftarrow BUS$

Aritméticas entre registros

1. $BUS \leftarrow R[rs] : R_a \leftarrow BUS$
2. $BUS \leftarrow R[r1] : R_b \leftarrow ALU[opCode]$
3. $BUS \leftarrow R_b : R[rs] \leftarrow BUS$

Aritmética registro-constante

1. $BUS \leftarrow R[rs] : R_a \leftarrow BUS$
2. $BUS \leftarrow R_{in} : R_b \leftarrow ALU[opCode]$
3. $BUS \leftarrow R_b : R[rs] \leftarrow BUS$

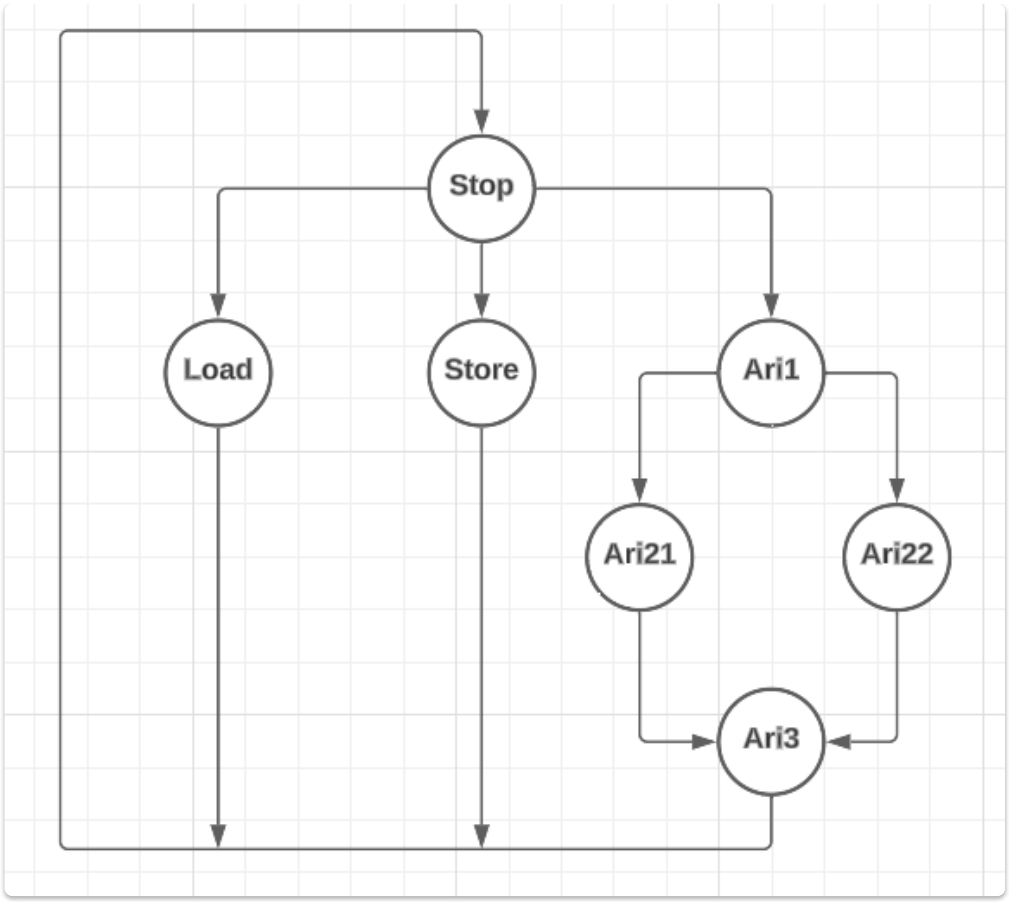
3.5) Diseño de la unidad de control

Implementaremos el control con una máquina de estados, donde tendremos un estado para cada conjunto de señales de control distintas. Por lo tanto, los estados serán los siguientes:

- STOP
- LOAD
- STORE
- ARI1
- ARI21
- ARI22
- ARI3

Los estados ARI1 y ARI3 son compartidos para las instrucciones aritméticas entre registros y las instrucciones aritméticas registro-constante, ya que comparten los mismos primer y tercer paso en el RTL concreto.

3.5.1) Transición de estados



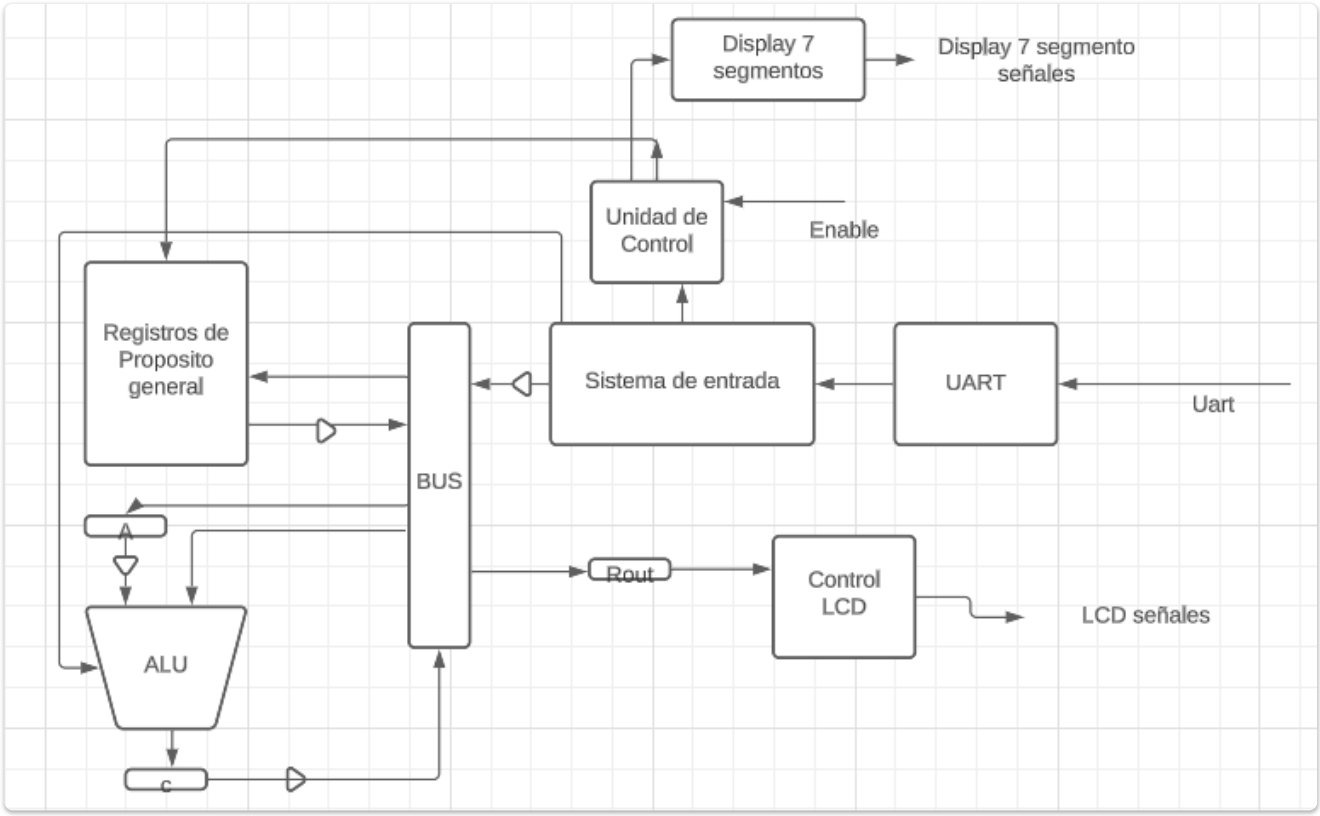
3.5.2) Valores de señales de control para cada estado

Estado	ControlBusIN	ControlBusB	ControlBusBanco	WriteBanco	WriteOUT	WriteA	WriteB
STOP	0	0	0	0	0	0	0
LOAD	1	0	0	1	0	0	0
STORE	0	0	1	0	1	0	0
ARI1	0	0	1	0	0	1	0
ARI21	0	0	1	0	0	0	1
ARI22	1	0	0	0	0	0	1
ARI3	0	1	0	1	0	0	0

3.6) Adiciones y observaciones de nuestro proyecto

Sistema de entrada: Se realizó un sistema de entrada que contiene el registro de entrada y una lógica con una máquina de estados para poder acomodar los datos ingresados por UART.
Display de 7 segmentos y unidad de control: Se configuró la unidad de control para que haga la transición en un pulso de reloj de 1 Hz, y a cada estado de la unidad de control se le asignó un número que se muestra en un bloque de display de 7 segmentos con fines didácticos.
Registro: Se implementó un único bloque de registro que cuenta con una salida TriState y se instanció varias veces.
Control del LCD: Se diseñó un control para el LCD con un contador interno y lógica, ya que los números almacenados en este micro son signed y es necesario convertirlos en ASCII para mostrarlos en el display.
Señales de control faltantes: Las señales de control que no se asignaron en la unidad de control se mantienen bajo control a lo largo de toda la ejecución, por lo que no fue necesario modificarlas.

3.7) Diagrama de bloque del proyecto completo



4) Implementacion de nuestros bloques en VHDL

En esta sección, omitiré los bloques que se realizaron durante el cursado de la materia, como el decodificador de 7 segmentos (display de 7 segmentos) y el bloque UART (aunque este último tiene modificaciones mínimas). También omitiré simulaciones de bloques demasiado sencillos.

4.1) Camino de datos

4.1.1) Registro de proposito general

language-vhdl

```vhdl
-- Bloque de registro de proposito general

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity BancoRegistros is
    port (
        clk: in std_logic;
        reset: in std_logic;
                triState: in std_logic;
        data_in: in std_logic_vector(31 downto 0);
        read_enable: in std_logic;
        write_enable: in std_logic;
        addres: in std_logic_vector(2 downto 0);
        data_out: out std_logic_vector(31 downto 0)
    );
end entity BancoRegistros;

architecture BancoRegistros_a of BancoRegistros is
    type RegisterArray is array(0 to 7) of std_logic_vector(31 downto 0);
    signal Regs: RegisterArray;
         signal address : natural range 0 to 7;
begin
        address <= to_integer(unsigned(addres));

    process (clk, write_enable,reset)
    begin
                if reset = '0' then
                        -- Reset values
                        Regs <= (others => (others => '0'));
        elsif rising_edge(clk) then
                        if write_enable = '1' then
                                Regs(address) <= data_in;
                        end if;
        end if;
    end process;

         process(clk, triState, read_enable)
         begin
                if rising_edge(clk) then
                        if(read_enable='1' and triState='1') then
                                data_out<= regs(address);
                        else
                                data_out<=(others=>'Z');
                        end if;
                end if;
         end process;

end architecture BancoRegistros_a;


-- TB del banco de registro de propositos general
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BancoRegistros_TB is
end entity BancoRegistros_TB;

architecture tb_arch of BancoRegistros_TB is
    -- Component declaration
```

```
    component BancoRegistros is
        port (
            clk: in std_logic;
            reset: in std_logic;
            triState: in std_logic;
            data_in: in std_logic_vector(31 downto 0);
            read_enable: in std_logic;
            write_enable: in std_logic;
            address: in std_logic_vector(2 downto 0);
            data_out: out std_logic_vector(31 downto 0)
        );
    end component;

    -- Signal declarations
    signal clks: std_logic := '0';
    signal resets: std_logic := '0';
    signal triStates: std_logic := '0';
    signal data_ins: std_logic_vector(31 downto 0) := (others => '0');
    signal read_enables: std_logic := '0';
    signal write_enables: std_logic := '0';
    signal addresss: std_logic_vector(2 downto 0) := "000";
    signal data_outs: std_logic_vector(31 downto 0);

begin
    -- Component instantiation
    dut: BancoRegistros
        port map (
            clk => clks,
            reset => resets,
            triState => triStates,
            data_in => data_ins,
            read_enable => read_enables,
            write_enable => write_enables,
            address => addresss,
            data_out => data_outs
        );


    -- Stimulus process
    stimulus_process: process
    begin
        -- Reset
        resets <= '0';
        wait for 10 ns;
        resets <= '1';

        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;
        clks <= '1';
        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;

        -- Write data to registers
        data_ins <= "00000000000000000000000000000001"; -- Data to be written
        write_enables <= '1'; -- Enable write
```

```
        addresss <= "000"; -- Address of the register to write
                wait for 10 ns;
                clks <= '0';
        wait for 10 ns;
        clks <= '1';
        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;

                  data_ins <= "10000000000000000000000000000001"; -- Data to be written
        write_enables <= '1'; -- Enable write
        addresss <= "111"; -- Addre
                wait for 10 ns;
                clks <= '0';
        wait for 10 ns;
        clks <= '1';
        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;

                assert data_outs = (others => 'Z') report "32bits output mal" severity
error;




        -- Read data from registers
        read_enables <= '1'; -- Enable read
        addresss <= "000"; -- Address of the register to read
                triStates <= '1'; -- Enable tri-state
        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;
        clks <= '1';
        wait for 10 ns;
                clks <= '0';
        wait for 10 ns;
                assert data_outs = "00000000000000000000000000000001" report "output mal"
severity error;

                  wait for 10 ns;
                  read_enables <= '0';
                  triStates <= '0';
                  wait for 10 ns;

                  read_enables <= '1'; -- Enable read
        addresss <= "111"; -- Address of the register to read
                  triStates <= '1'; -- Enable tri-state
        wait for 10 ns;
                  clks <= '0';
        wait for 10 ns;
        clks <= '1';
        wait for 10 ns;
                  clks <= '0';
        wait for 10 ns;
                  assert data_outs = "10000000000000000000000000000001" report "output mal"
severity error;
```

```
        -- End simulation
        wait;
    end process;

end architecture tb_arch;
```

4.1.2) ALU

language-vhdl

```
-- ALU

entity ALU is
    port (
        ALUctl : in  std_logic_vector(1 downto 0);
        A      : in  std_logic_vector(31 downto 0);
        B      : in  std_logic_vector(31 downto 0);
        ALUOut : out std_logic_vector(31 downto 0);
        Zero   : out std_logic
    );
end entity ALU;

architecture ALU_a of ALU is
begin
    process (ALUctl, A, B)
    begin
        case ALUctl is
            when "00" =>
                ALUOut <= A and B;
            when "01" =>
                ALUOut <= A or B;
            when "10" =>
                ALUOut <=std_logic_vector(signed(A) + signed(B));
            when "11" =>
                ALUOut <=std_logic_vector(signed(A) - signed(B));
            when others =>
                ALUOut <= (others => '0');
        end case;

        Zero <= '1';
    end process;
end architecture ALU_a;

-- TB ALU

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU_TB is
end entity ALU_TB;

architecture tb_arch of ALU_TB is
    -- Component declaration
    component ALU is
```

```
        port (
            ALUctl : in  std_logic_vector(1 downto 0);
            A      : in  std_logic_vector(31 downto 0);
            B      : in  std_logic_vector(31 downto 0);
            ALUOut : out std_logic_vector(31 downto 0);
            Zero   : out std_logic
        );
    end component;

    -- Signal declarations
    signal ALUctl: std_logic_vector(1 downto 0) := "00";
    signal A: std_logic_vector(31 downto 0) := (others => '0');
    signal B: std_logic_vector(31 downto 0) := (others => '0');
    signal ALUOut: std_logic_vector(31 downto 0);
    signal Zero: std_logic;

begin
    -- Component instantiation
    dut: ALU
        port map (
            ALUctl => ALUctl,
            A => A,
            B => B,
            ALUOut => ALUOut,
            Zero => Zero
        );

    -- Stimulus process
    stimulus_process: process
    begin
        -- Test case 1: AND operation
        ALUctl <= "00"; -- AND operation
        A <= "00101010101010101010101010101010";
        B <= "01110000111100001111000011110000";
        wait for 10 ns;
                  assert (ALUOut= ("00101010101010101010101010101010" and
"01110000111100001111000011110000")) report "32bits output mal 1" severity error;
                  wait for 10 ns;


        -- Test case 2: OR operation
        ALUctl <= "01"; -- OR operation
        wait for 10 ns;
                  assert (ALUOut= ("00101010101010101010101010101010" or
"01110000111100001111000011110000")) report "32bits output mal 2" severity error;
                  wait for 10 ns;

        -- Test case 3: Addition operation
        ALUctl <= "10"; -- Addition operation
        wait for 10 ns;
                  assert (ALUOut = "10011011100110111001101110011010") report "32bits
output mal 3" severity error;
                  wait for 10 ns;

        -- Test case 4: Subtraction operation
                  A <= "00000000000000000000000000000010";
        B <= "00000000000000000000000000000111";
```

```
        ALUctl <= "11"; -- Subtraction operation
        wait for 10 ns;
                  assert (ALUOut=  "11111111111111111111111111111011") report "32bits
output mal 4" severity error;
                  wait for 10 ns;

        -- End simulation
        wait;
    end process;

end architecture tb_arch;
```

4.1.3)Registro

language-vhdl

```
-- Bloque Registro
library ieee;
use ieee.std_logic_1164.all;

entity Register32bit is
    port (
        clk: in std_logic;
        reset: in std_logic;
                  triState: in std_logic;
        data_in: in std_logic_vector(31 downto 0);
        read_enable: in std_logic;
        write_enable: in std_logic;
        data_out: out std_logic_vector(31 downto 0)
    );
end entity Register32bit;

architecture Register32bit_a of Register32bit is
    signal reg: std_logic_vector(31 downto 0);
begin
    process (clk,reset)
    begin
                if reset = '0' then
                        -- Reset value
                        reg <= (others => '0');
        elsif rising_edge(clk) then
                        if write_enable = '1' then
                                reg <= data_in;
                        end if;
        end if;
    end process;

         process(clk, triState, read_enable)
         begin
                if rising_edge(clk) then
                        if(read_enable='1' and triState='1') then
                                data_out<= reg;
                        else
                                data_out<=(others=>'Z');
                        end if;
                end if;
         end process;
```

```vhdl
end architecture Register32bit_a;


-- TB del registro

library ieee;
use ieee.std_logic_1164.all;

entity Register32bit_TB is
end entity Register32bit_TB;

architecture tb_arch of Register32bit_TB is
    -- Component declaration
    component Register32bit is
        port (
            clk: in std_logic;
            reset: in std_logic;
            triState: in std_logic;
            data_in: in std_logic_vector(31 downto 0);
            read_enable: in std_logic;
            write_enable: in std_logic;
            data_out: out std_logic_vector(31 downto 0)
        );
    end component;

    -- Signal declarations
    signal clk: std_logic := '0';
    signal reset: std_logic := '0';
    signal triState: std_logic := '0';
    signal data_in: std_logic_vector(31 downto 0) := (others => '0');
    signal read_enable: std_logic := '0';
    signal write_enable: std_logic := '0';
    signal data_out: std_logic_vector(31 downto 0);

begin
    -- Component instantiation
    dut: Register32bit
        port map (
            clk => clk,
            reset => reset,
            triState => triState,
            data_in => data_in,
            read_enable => read_enable,
            write_enable => write_enable,
            data_out => data_out
        );

    -- Clock generation process
    clk_process: process
    begin
        while now < 100 ns loop
            clk <= '0';
            wait for 5 ns;
            clk <= '1';
            wait for 5 ns;
        end loop;
        wait;
```

```
    end process;

    -- Stimulus process
    stimulus_process: process
    begin
        -- Reset
        reset <= '0';
        wait for 10 ns;
        reset <= '1';
        wait for 10 ns;

        -- Write data to register
        data_in <= "00000000000000000000000000000001"; -- Data to be written
        write_enable <= '1'; -- Enable write
        wait for 10 ns;
                  write_enable <= '0';
                  wait for 10 ns;
        -- Read data from register
        read_enable <= '1'; -- Enable read
                  triState <= '1'; -- Enable tri-state
        wait for 10 ns;
                  assert data_out = "00000000000000000000000000000001" report "output mal"
severity error;
                  wait for 10 ns;
                  read_enable<='0';
                  write_enable<='1';
                  triState<='0';


                   -- Write data to register
        data_in <= "11000000000000000000000000000001"; -- Data to be written
        write_enable <= '1'; -- Enable write
        wait for 10 ns;
                  write_enable <= '0';
                  wait for 10 ns;
        -- Read data from register
        read_enable <= '1'; -- Enable read
                  triState <= '1'; -- Enable tri-state
        wait for 10 ns;
                  assert data_out = "11000000000000000000000000000001" report "output mal"
severity error;



        -- End simulation
        wait;
    end process;

end architecture tb_arch;
```

4.1.4) BUS

Aunque no tengo un bloque específico de BUS de comunicación, me pareció importante hacer una mención especial sobre cómo lo utilicé en mi diseño.

El BUS en mi diseño es una señal intermedia a la cual conecté todas las entradas o salidas de datos de 32 bits de los diferentes componentes. Para garantizar un uso exclusivo del BUS por parte de cada componente, agregué buffers TriState en las salidas de estos componentes y configuré la unidad de control para gestionar adecuadamente el acceso al BUS. De esta manera, logré una comunicación sincronizada y sin interferencias.

4.2)Unidad de control

```vhdl
-- Unidad de control
library ieee;
use ieee.std_logic_1164.all;

entity UnidadControl is
    port (
        enable : in std_logic;
        opcode : in std_logic_vector(3 downto 0);
                OpReg: in std_logic_vector(5 downto 0);
        clk : in std_logic;
        reset : in std_logic;
        ControlBusIN, ControlBusB, ControlBusBanco : out std_logic;
                WriteBanco, WriteOUT, WriteA, WriteB : out std_logic;
                ReadBanco, ReadA, ReadB : out std_logic;
        Segmentos : out std_logic_vector(3 downto 0):="0000";
                Addres : out std_logic_vector(2 downto 0)
    );
end entity UnidadControl;

architecture UnidadControl_a of UnidadControl is
    type StateType is (Stop, Load, Store, Ari1, Ari21, Ari22, Ari3);
    signal currentState, nextState : StateType;

begin
    process (clk, reset)
    begin
        if reset = '0' then
            currentState <= Stop;
        elsif rising_edge(clk) then
            currentState <= nextState;
        end if;
    end process;

    process (currentState, enable, opcode,clk,OpReg)
    begin
        case currentState is
            when Stop =>
                ControlBusIN <= '0';
                ControlBusB <= '0';
                ControlBusBanco <= '0';
                WriteBanco <= '0';
                WriteOUT <= '0';
                WriteA <= '0';
                WriteB <= '0';
                ReadBanco <= '0';
                                        Segmentos <="0000";
                ReadA <= '0';
```

```
            ReadB <= '0';
            Addres <= "000";

            if enable = '0' then
                if opcode = "1001" then
                    nextState <= Store;
                elsif opcode = "1111" then
                    nextState <= Load;
                elsif opcode(3) = '0' then
                    nextState <= Ari1;
                else
                    nextState <= Stop;
                end if;
            else
                nextState <= Stop;
            end if;

        when Load =>
            ControlBusIN <= '1';
            ControlBusB <= '0';
            ControlBusBanco <= '0';
            WriteBanco <= '1';
            WriteOUT <= '0';
            WriteA <= '0';
            WriteB <= '0';
            ReadBanco <= '0';
            Segmentos <="0001";
            ReadA <= '0';
            ReadB <= '0';
            Addres <= OpReg(2 downto 0);
            nextState <= Stop;

        when Store =>
            ControlBusIN <= '0';
            ControlBusB <= '0';
            ControlBusBanco <= '1';
            WriteBanco <= '0';
            WriteOUT <= '1';
            WriteA <= '0';
            WriteB <= '0';
            ReadBanco <= '1';
            Segmentos <="0010";
            ReadA <= '0';
            ReadB <= '0';
            Addres <= OpReg(2 downto 0);
            nextState <= Stop;

        when Ari1 =>
            ControlBusIN <= '0';
            ControlBusB <= '0';
            ControlBusBanco <= '1';
            WriteBanco <= '0';
            WriteOUT <= '0';
            WriteA <= '1';
            WriteB <= '0';
            ReadBanco <= '1';
                                    Segmentos <="0011";
```

```
                ReadA <= '0';
                ReadB <= '0';
                Addres <= OpReg(2 downto 0);

                if opcode(2) = '1' then
                    nextState <= Ari22;
                else
                    nextState <= Ari21;
                end if;

            when Ari22 =>
                ControlBusIN <= '1';
                ControlBusB <= '0';
                ControlBusBanco <= '0';
                WriteBanco <= '0';
                WriteOUT <= '0';
                WriteA <= '0';
                WriteB <= '1';
                ReadBanco <= '0';
                Segmentos <="0100";
                ReadA <= '0';
                ReadB <= '0';
                Addres <= "000";
                nextState <= Ari3;

            when Ari21 =>
                ControlBusIN <= '0';
                ControlBusB <= '0';
                ControlBusBanco <= '1';
                WriteBanco <= '0';
                WriteOUT <= '0';
                WriteA <= '0';
                WriteB <= '1';
                ReadBanco <= '1';
                Segmentos <="0101";
                ReadA <= '0';
                ReadB <= '0';
                Addres <= OpReg(5 downto 3);
                nextState <= Ari3;

            when Ari3 =>
                ControlBusIN <= '0';
                ControlBusB <= '1';
                ControlBusBanco <= '0';
                WriteBanco <= '1';
                WriteOUT <= '0';
                                    Segmentos <="0110";
                WriteA <= '0';
                WriteB <= '0';
                ReadBanco <= '0';
                ReadA <= '0';
                ReadB <= '1';
                Addres <= OpReg(2 downto 0);
                nextState <= Stop;
        end case;
    end process;
```

```
end architecture UnidadControl_a;

-- TB de la unidad de control

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity UnidadControl_TB is
end entity UnidadControl_TB;

architecture Behavioral of UnidadControl_TB is
    -- Component declaration for the unit under test (UUT)
    component UnidadControl is
        port (
            enable : in std_logic;
            opcode : in std_logic_vector(3 downto 0);
            OpReg : in std_logic_vector(5 downto 0);
            clk : in std_logic;
            reset : in std_logic;
            ControlBusIN, ControlBusB, ControlBusBanco : out std_logic;
            WriteBanco, WriteOUT, WriteA, WriteB : out std_logic;
            ReadBanco, ReadA, ReadB : out std_logic;
            Addres : out std_logic_vector(2 downto 0)
        );
    end component;

    -- Signal declarations
    signal enable_sig : std_logic := '0';
    signal opcode_sig : std_logic_vector(3 downto 0) := (others => '0');
    signal OpReg_sig : std_logic_vector(5 downto 0) := (others => '0');
    signal clk_sig : std_logic := '0';
    signal reset_sig : std_logic := '0';
    signal ControlBusIN_sig, ControlBusB_sig, ControlBusBanco_sig : std_logic;
    signal WriteBanco_sig, WriteOUT_sig, WriteA_sig, WriteB_sig : std_logic;
    signal ReadBanco_sig, ReadA_sig, ReadB_sig : std_logic;
    signal Addres_sig : std_logic_vector(2 downto 0);

begin
    -- Instantiate the unit under test (UUT)
    uut : UnidadControl
        port map (
            enable => enable_sig,
            opcode => opcode_sig,
            OpReg => OpReg_sig,
            clk => clk_sig,
            reset => reset_sig,
            ControlBusIN => ControlBusIN_sig,
            ControlBusB => ControlBusB_sig,
            ControlBusBanco => ControlBusBanco_sig,
            WriteBanco => WriteBanco_sig,
            WriteOUT => WriteOUT_sig,
            WriteA => WriteA_sig,
            WriteB => WriteB_sig,
            ReadBanco => ReadBanco_sig,
            ReadA => ReadA_sig,
            ReadB => ReadB_sig,
```

```
            Addres => Addres_sig
        );




    -- Stimulus process
    stim_proc: process
    begin
                reset_sig <= '0';
                wait for 10 ns;
        reset_sig <= '1';
                wait for 10 ns;


        -- Test case LOAD
                opcode_sig <= "1000";
                OpReg_sig <= "000001";
                wait for 10 ns;
        enable_sig <= '1';
                wait for 10 ns;
                clk_sig <= '1';
        wait for 10 ns;
        clk_sig <= '0';
                wait for 10 ns;
                assert ControlBusIN_sig = '1' report "FalloControlBus1" severity error;
                assert Addres_sig = "001" report "FalloRegistro1" severity error;
                enable_sig<='0';
        clk_sig <= '0';
        wait for 10 ns;
        clk_sig <= '1';
        wait for 10 ns;
        clk_sig <= '0';
        wait for 10 ns;


        -- Test case STORE
        opcode_sig <= "1001";
        OpReg_sig <= "000010";
                wait for 10 ns;
                enable_sig <= '1';
                wait for 10 ns;
                clk_sig <= '1';
        wait for 10 ns;
        clk_sig <= '0';
                wait for 10 ns;
                assert ControlBusBanco_sig= '1' report "FalloControlBus2" severity error;
                assert Addres_sig = "010" report "FalloRegistro2" severity error;
                enable_sig <= '0';


        clk_sig <= '0';
        wait for 10 ns;
        clk_sig <= '1';
        wait for 10 ns;
        clk_sig <= '0';
        wait for 10 ns;
```

```
        -- Test case ARI
                                -- ARI1
        opcode_sig <= "0000";
        OpReg_sig <= "000010";
                  wait for 10 ns;
                  enable_sig <= '1';
                  wait for 10 ns;
                  clk_sig <= '1';
        wait for 10 ns;
        clk_sig <= '0';
                  wait for 10 ns;

                  assert ControlBusBanco_sig= '1' report "FalloControlBus3" severity error;
                  assert Addres_sig = "010" report "FalloRegistro3" severity error;


                  wait for 10 ns;
        clk_sig <= '0';
        wait for 10 ns;
        clk_sig <= '1';
                  wait for 10 ns;
                  clk_sig <= '0';
        wait for 10 ns;

                                -- ARI21
                  assert ControlBusBanco_sig= '1' report "FalloControlBus4" severity error;
                  assert Addres_sig = "000" report "FalloRegistro4" severity error;



        wait for 10 ns;
        clk_sig <= '0';
        wait for 10 ns;
                  clk_sig <= '1';
                  wait for 10 ns;
                  clk_sig <= '0';
        wait for 10 ns;


                                -- ARI3
                  assert ControlBusB_sig= '1' report "FalloControlBus5" severity error;
                  assert Addres_sig = "010" report "FalloRegistro5" severity error;

                  wait for 10 ns;

        wait;
    end process;

end architecture Behavioral;
```

4.3) Control LCD

```
library IEEE;                                                    language-vhdl
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;

entity LCD_Disp is
port( LCD_DataIn  : IN  STD_LOGIC_VECTOR(12 downto 0);
                Clk              : IN  STD_LOGIC := '0';
                LCD_DataOut : OUT STD_LOGIC_VECTOR(7 downto 0);
                LCD_E           : OUT STD_LOGIC;
      LCD_RS     : OUT STD_LOGIC);
end entity;


architecture LCD_Disp_arc of LCD_Disp is

Type arrayData is array (0 to 12) of STD_LOGIC_VECTOR(7 downto 0);

signal Datas: arrayData;
signal TempVal   : INTEGER;
signal TempVal_1 : INTEGER range 0 to 9:= 1;
signal TempVal_2 : INTEGER range 0 to 9:= 1;
signal TempVal_3 : INTEGER range 0 to 9:= 1;
signal TempVal_4 : INTEGER range 0 to 9:= 1;
signal TempVal_signal : std_logic;
begin

--Commands--

Datas(0)  <= X"38";
Datas(1)  <= X"0c";
Datas(2)  <= X"06";
Datas(3)  <= X"80";

--Datas--

Datas(4)  <= x"52";  --R
Datas(5)  <= x"73";  --s
Datas(6)  <= x"3D";     --=
Datas(7)  <= x"20";     -- ESPACIO


TempVal   <= (to_integer(UNSIGNED(LCD_DataIn)));    -- FORMULA PARA LA CONVERSION
TempVal_1 <= (TempVal) mod 10;         -- UNIDAD
TempVal_2 <= (TempVal/10) mod 10;               -- DECIMA
TempVal_3 <= (TempVal/100) mod 10;              -- CENTESIMA
TempVal_4 <= (TempVal/1000)mod 10;              -- MILESIMA
TempVal_signal <= LCD_DataIn(12);



with (TempVal_1) select
Datas(12) <=  x"30" when 0,
              x"31" when 1,
                                x"32" when 2,
                                x"33" when 3,
                                x"34" when 4,
                                x"35" when 5,
                                x"36" when 6,
                                x"37" when 7,
```

```
                                x"38" when 8,
                                x"39" when 9,
                                x"30" when others;



with (TempVal_2) select
Datas(11) <=  x"30" when 0,
              x"31" when 1,
                                x"32" when 2,
                                x"33" when 3,
                                x"34" when 4,
                                x"35" when 5,
                                x"36" when 6,
                                x"37" when 7,
                                x"38" when 8,
                                x"39" when 9,
                                x"30" when others;


with (TempVal_3) select
Datas(10) <=  x"30" when 0,
              x"31" when 1,
                                x"32" when 2,
                                x"33" when 3,
                                x"34" when 4,
                                x"35" when 5,
                                x"36" when 6,
                                x"37" when 7,
                                x"38" when 8,
                                x"39" when 9,
                                x"30" when others;

with (TempVal_4) select
Datas(9) <=  x"30" when 0,
              x"31" when 1,
                                x"32" when 2,
                                x"33" when 3,
                                x"34" when 4,
                                x"35" when 5,
                                x"36" when 6,
                                x"37" when 7,
                                x"38" when 8,
                                x"39" when 9,
                                x"30" when others;


with (TempVal_Signal) select
Datas(8) <=  x"2B" when '0',
              x"2D" when '1',
                                x"30" when others;



LCD_proc: process(Clk)

       variable i : integer := 0;
```

```
        variable j : integer range 0 to 32 := 0;

    begin
        if (Clk'event and Clk = '1') then
           if(i <= 85000) then
                                        i := i + 1;
                                        LCD_E <= '1';
                                        if(j<=12) then
                                                LCD_DataOut <= DataS(j)(7 downto 0);
                                        else
                                                LCD_DataOut <= x"20";
                                        end if;
           elsif(i > 85000 and i < 160000) then i := i + 1; lcd_e <= '0';
           elsif(i = 160000) then j := j + 1; i := 0;
           end if;

           if(j < 4) then LCD_RS <= '0';                                                -
- Command Signal --
      elsif (j >= 4 and j <= 32) then lcd_rs <= '1';    -- Data Signal --
      end if;

           if(j >= 32) then j := 0;                -- Repeat Data Display Routine --
           end if;
        end if;
    end process LCD_proc;

end LCD_Disp_arc;
```

4.4) Sistema de entrada

language-vhdl

```vhdl
-- Sistema de entrada
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity InputSystem is
    port (
        clk: in std_logic;
        reset: in std_logic;
        enable: in std_logic;
                  rx_done: in std_logic;
        input_data: in std_logic_vector(7 downto 0);
        output_data_ALU: out std_logic_vector(1 downto 0);
        --output_data_OpCode: out std_logic_vector(3 downto 0);
                  output_data_Reg : out std_logic_vector(5 downto 0);
        output_data_32bits: out std_logic_vector(31 downto 0)
    );
end entity InputSystem;

architecture InputSystem_a of InputSystem is
    signal register_data: std_logic_vector(15 downto 0):="0000000000000000"; --Nuevo
    signal state: integer range 0 to 1 := 0;
begin

         output_data_ALU <= register_data(1 downto 0);
    --output_data_OpCode <= register_data(3 downto 0);
```

```
    output_data_Reg(2 downto 0) <= register_data(10 downto 8);
         output_data_Reg(5 downto 3) <= register_data(14 downto 12);

    process (reset, rx_done)
    begin
                if reset = '0' then
                        -- Reset values
                        register_data <= "0000000000000000";
            state <= 0;
        elsif rx_done='1' then
                                -- State machine
            case state is
                when 0 =>
                    register_data(7 downto 0) <= input_data;
                    state <= 1;
                when 1 =>
                    register_data(15 downto 8) <= input_data;
                    state <= 0;
            end case;
        end if;
    end process;



    process (enable, register_data, clk)
    begin
        if enable = '1' then
            output_data_32bits(7 downto 0) <= register_data(7 downto 0);
                output_data_32bits(31 downto 8) <=(others => '0');
        else
            output_data_32bits <= (others => 'Z');
        end if;
    end process;

end architecture InputSystem_a ;



-- TB sistema de entrada

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity InputSystem_TB is
end entity InputSystem_TB;


architecture InputSystem_TB_arch of InputSystem_TB is
    -- Component declaration for the DUT (Design Under Test)
    component InputSystem
        port (
            clk: in std_logic;
            reset: in std_logic;
            enable: in std_logic;
            rx_done: in std_logic;
            input_data: in std_logic_vector(7 downto 0);
```

```
            output_data_ALU: out std_logic_vector(1 downto 0);
            output_data_OpCode: out std_logic_vector(3 downto 0);
            output_data_Reg : out std_logic_vector(5 downto 0);
            output_data_32bits: out std_logic_vector(31 downto 0)
        );
    end component;

    -- Signals for testbench stimulus
    signal tb_clk: std_logic := '0';
    signal tb_reset: std_logic := '1';
    signal tb_enable: std_logic;
    signal tb_rx_done: std_logic;
    signal tb_input_data: std_logic_vector(7 downto 0);

    -- Signals for capturing DUT outputs
    signal tb_output_data_ALU: std_logic_vector(1 downto 0);
    signal tb_output_data_OpCode: std_logic_vector(3 downto 0);
    signal tb_output_data_Reg: std_logic_vector(5 downto 0);
    signal tb_output_data_32bits: std_logic_vector(31 downto 0);

begin
    -- Instantiate the DUT
    uut: InputSystem port map (
            clk => tb_clk,
            reset => tb_reset,
            enable => tb_enable,
            rx_done => tb_rx_done,
            input_data => tb_input_data,
            output_data_ALU => tb_output_data_ALU,
            output_data_OpCode => tb_output_data_OpCode,
            output_data_Reg => tb_output_data_Reg,
            output_data_32bits => tb_output_data_32bits
        );


         tb_clk <= not tb_clk after 5 ns;

    -- Stimulus process
    stimulus: process
    begin
        tb_reset <= '0';
        tb_enable <= '0';
        tb_rx_done <= '0';
        tb_input_data <= (others => '0');
        wait for 10 ns;
        tb_reset <= '1';
        wait for 10 ns;

        -- Initialize inputs
        tb_enable <= '0';
        tb_rx_done <= '0';
        tb_input_data <= (others => '0');

        -- Wait a few clock cycles
        wait for 10 ns;

        -- Generate stimulus
```

```
        tb_input_data <= "11111111"; -- First input value
        wait for 10 ns;
        tb_rx_done <= '1';
        wait for 10 ns;
        tb_rx_done <= '0';
        wait for 10 ns;
        tb_input_data <= "00000000";
                  tb_enable <= '1';
        wait for 10 ns;
        tb_rx_done <= '1';
        wait for 10 ns;
        tb_rx_done <= '0';
        wait for 10 ns;

        -- Check outputs
        assert tb_output_data_ALU = "11" report "ALU1 output incorrect" severity error;
        assert tb_output_data_OpCode = "1111" report "OpCode1 output incorrect" severity
error;
        assert tb_output_data_Reg = "001111"report "Reg1 output incorrect" severity error;
        assert tb_output_data_32bits = "00000000000000000000000000000000" report "32bits1
output incorrect" severity error;

        tb_input_data <= "01010101";
        wait for 10 ns;
        tb_rx_done <= '1';
        wait for 10 ns;
        tb_rx_done <= '0';
        wait for 10 ns;
        tb_input_data <= "00000000";
        wait for 10 ns;
        tb_rx_done <= '1';
        wait for 10 ns;
        tb_rx_done <= '0';
        wait for 10 ns;

        tb_enable <= '1'; -- enable data storage
        wait for 10 ns;

        -- Check outputs
        assert tb_output_data_ALU = "01" report "ALU2 output incorrect" severity error;
        assert tb_output_data_OpCode = "0101" report "OpCode2 output incorrect" severity
error;
        assert tb_output_data_Reg = "000101" report "Reg2 output incorrect" severity error;
        assert tb_output_data_32bits = "00000000000000000000000000000000" report "32bits2
output incorrect" severity error;

        wait;
    end process;

end architecture InputSystem_TB_arch;
```

5) Funcionamiento

1. Apretar reset

2. Mandar datos por UART
3. Seleccionar la instruccion deseada
4. Habilitar el funcionamiento del micro
    - Esto se hace apretando KEY0

5.1) Tablas de OpCode

Instruccion	OpCode
LOAD	1111
STORE	1001
AND Rs y R1	0000
OR Rs y R1	0001
ADD Rs y R1	0010
SUB Rs y R1	0011
AND Rs y cte	0100
OR Rs y cte	0101
ADD Rs y cte	0110
SUB Rs y cte	0111

5.2) Metodo para mandar por UART

- Se debe seleccionar para mandar los datos en HEXA
- Enviar 2 datos de 8 bits, uno hace referencia a los 2 registros seleccionados y el otro dato hace referencia a la constante

6) Observaciones

6.1) Bloques obviados en el informe

Es muy importante tener en cuenta que para el correcto funcionamiento de este proyecto se requirieron otros bloques que no fueron incluidos en el informe. A continuación, los mencionaré junto con su uso:

- Contador: Es necesario para la comunicación UART, así como para generar la frecuencia de reloj de la unidad de control a 1Hz.
- Antirrebote: Se utilizó para evitar problemas de rebote en el botón de la placa, el cual se usa para habilitar al CPU a ejecutar una instrucción.

7) Conclusiones

7.1) Logros alcanzados

Desde siempre me he interesado mucho en el desarrollo de hardware de este tipo. Hace unos meses, creía que no sería capaz de realizar un proyecto como este debido a mis conocimientos limitados y a la dificultad que percibía en estos temas. Sin embargo, la realización de este proyecto ha sido un gran logro

personal para mí. En términos de conocimientos, me ha permitido profundizar en los conceptos de VHDL y su implementación en FPGA, además de ampliar mi perspectiva hacia un nuevo mundo de aprendizaje.

7.2) Recomendaciones para trabajos futuros

Recomiendo a las personas que planeen realizar un proyecto similar que utilicen todos los sistemas de entrada y salida disponibles en la FPGA para depurar posibles errores. Es muy difícil hacerlo únicamente con simulaciones de banco de pruebas. En mi caso, el uso del display de 7 segmentos y los interruptores de la placa fueron de gran ayuda cuando enfrenté problemas en la comunicación por UART.

8)Referencias

- Cátedra de Arquitectura de Computadoras
- Cátedra de Sistemas con Microprocesadores.
- Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design RISC-V Edition: The Hardware Software Interface.
- Tocci, R. J. (Ed.). (n.d.). Sistemas Digitales (7.ª edición).

9)ANEXOS

9.1)Test-Driven Development TDD

El desarrollo basado en pruebas (Test-Driven Development, TDD) es una práctica de desarrollo de software que se basa en escribir pruebas automatizadas antes de escribir el código de implementación. Es un enfoque iterativo e incremental en el que las pruebas unitarias impulsan el diseño y desarrollo del software.

El proceso de desarrollo basado en pruebas sigue generalmente estos pasos:

1. Escribir una prueba: Se escribe una prueba automatizada que describe el comportamiento esperado del código. Esta prueba inicialmente fallará porque el código aún no ha sido implementado.
2. Ejecutar la prueba: Se ejecuta la prueba automatizada y se verifica que falle, lo cual es esperado en este punto.
3. Escribir el código de implementación mínimo: Se implementa la cantidad mínima de código necesaria para que la prueba pase.
4. Ejecutar todas las pruebas: Se ejecutan todas las pruebas automatizadas, incluyendo la que se acaba de implementar. Se verifica que todas pasen correctamente.
5. Refactorizar el código: Se mejora la estructura y calidad del código sin cambiar su comportamiento. Durante este paso, se pueden identificar nuevas pruebas o mejoras en las existentes.
6. Repetir el ciclo: Se repiten los pasos anteriores para desarrollar nuevas características o modificar las existentes.

En VHDL, el enfoque de desarrollo basado en pruebas (TDD) se puede implementar utilizando las sentencias `assert` para escribir las pruebas automatizadas

9.2)Procesador Multiciclo

En verilog

```language-verilog
module RISCVCPU (clock);
// Instruction opcodes
        parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
input clock;
reg [31:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut, MEMWBValue;
reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
                IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; // Access register fields
wire [6:0] IFIDop, IDEXop, EXMEMop, MEMWBop; // Access opcodes
wire [31:0] Ain, Bin; // the ALU inputs
// declare the bypass signals
wire bypassAfromMEM, bypassAfromALUinWB,
        bypassBfromMEM, bypassBfromALUinWB,
        bypassAfromLDinWB, bypassBfromLDinWB;
wire stall; // stall signal
wire takebranch;
assign IFIDop = IFIDIR[6:0];
assign IFIDrs1 = IFIDIR[19:15];
assign IFIDrs2 = IFIDIR[24:20];
assign IDEXop = IDEXIR[6:0];
assign IDEXrs1 = IDEXIR[19:15];
assign IDEXrs2 = IDEXIR[24:20];
assign EXMEMop = EXMEMIR[6:0];
assign EXMEMrd = EXMEMIR[11:7];
assign MEMWBop = MEMWBIR[6:0];
assign MEMWBrd = MEMWBIR[11:7];
// The bypass to input A from the MEM stage for an ALU operation
assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) && (EXMEMop == ALUop);
// The bypass to input B from the MEM stage for an ALU operation
assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) && (EXMEMop == ALUop);
// The bypass to input A from the WB stage for an ALU operation
assign bypas sAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) && (MEMWBop == ALUop);
// The bypass to input B from the WB stage for an ALU operation
assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) && (MEMWBop == ALUop);
// The bypass to input A from the WB stage for an LW operation
assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) && (MEMWBop == LW);
// The bypass to input B from the WB stage for an LW operation
assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEX rs2 != 0) && (MEMWBop == LW);
// The A input to the ALU is bypassed from MEM if there is a bypass there,
// Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
assign Ain = bypassAfromMEM ? EXMEMALUOut :
                        (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;
// The B input to the ALU is bypassed from MEM if there is a bypass there,
// Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
assign Bin = bypassBfromMEM ? EXMEMALUOut :
                        (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
IDEXB;
// The signal for detecting a stall based on the use of a result from LW
assign stall = (MEMWBop == LW) && ( // source instruction is a load
                                (((IDEXop == LW) || (IDEXop == SW)) && (IDEXrs1 ==
MEMWBrd))|| // stall for address calc
                                ((IDEXop == ALUop) && ((IDEXrs1 == MEMWBrd) || (IDEXrs2 ==
MEMWBrd)))); // ALU use
// Signal for a taken branch: instruction is BEQ and registers are equal
```

```
assign takebranch = (IFIDop == BEQ) && (Regs[IFIDrs1] == Regs[IFIDrs2]);
integer i; // used to initialize registers
initial
begin
        PC = 0;
        IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs in pipeline
registers
        for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so they aren't
cares
        end
// Remember that ALL these actions happen every pipe stage and with the use of <= they
happen in parallel!
        always @(posedge clock)
        begin
                if (~stall)
                begin // the first three pipeline stages stall if there is a load hazard
                        if (~takebranch)
                        begin // first instruction in the pipeline is being fetched
normally
                                IFIDIR <= IMemory[PC >> 2];
                                PC <= PC + 4;
                        end
                        else
                        begin // a taken branch is in ID; instruction in IF is wrong;
insert a NOP and reset the PC
                                IFIDIR <= NOP;
                                PC <= PC + {{52{IFIDIR[31]}}, IFIDIR[7], IFIDIR[30:25],
IFIDIR[11:8], 1'b0};
                        end
// second instruction in pipeline is fetching registers
                        IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two
registers
                        IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!
                        // third instruction is doing addre ss calculation or ALU operation
                        if (IDEXop == LW)
                                EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
                        else if (IDEXop == SW)
                                EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
                        else if (IDEXop == ALUop)
                                case (IDEXIR[31:25]) // case for the various R-type
instructions
                                        0: EXMEMALUOut <= Ain + Bin; // add operation
                                        default: ; // other R-type operations: subtract,
SLT, etc.
                                endcase
                                EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR &
B register
                        end
                        else EXMEMIR <= NOP; // Freeze first three stages of pipeline;
inject a nop into the EX output
// Mem stage of pipeline
                        if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
                        else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
                        else if (EXMEMop == SW) DMemory[EXMEMALUOut >> 2] <= EXMEMB;
```

```
//store
                        MEMWBIR <= EXMEMIR; // pass along IR

                        // WB stage
                        if (((MEMWBop == LW) || (MEMWBop == ALUop)) && (MEM WBrd != 0)) //
update registers if load/ALU operation and destination not 0
                        Regs[MEMWBrd] <= MEMWBValue;
                end
endmodule
```