Título de la práctica:	Simulación de lenguaje ensamblador en programación orientada a objetos
Unidad de Aprendizaje:	Paradigmas de programación
Nombre de los alumnos:	Castillo Molano Sergio de Jesus, Flores Mora Juan Carlos, Flores Sotelo Miguel Alejandro

Desarrollo de la práctica

Se implementará la práctica del ciclo fetch, pero ahora con otro paradigma de programación que es “programación Orientada a Objetos”.

La simulación del **ciclo fetch** de una computadora es esencial para comprender cómo funciona internamente un procesador. Este ciclo, llevado a cabo por la unidad de control de la CPU, implica una secuencia de pasos para recuperar, decodificar y ejecutar instrucciones almacenadas en memoria. Aquí detallamos los pasos necesarios para implementar esta simulación utilizando un conjunto de instrucciones representadas en código binario, las cuales serán leídas desde un archivo de texto.

Lectura de Instrucciones desde el Archivo de Texto:

Implementaremos un programa en Python que lea las instrucciones desde un archivo de texto en formato binario.

Cada línea del archivo representará una instrucción, donde los primeros tres números binarios corresponden a la operación a realizar y los últimos dígitos o registros indican a qué registros de memoria se accederá.

Clase Instrucciones:

Creamos la clase Instrucciones para encapsular el comportamiento relacionado con el manejo y ejecución de instrucciones.

En el método __init__, inicializamos los atributos de la clase, como la lista de instrucciones, el valor inicial del acumulador y el índice para la posición actual.

Definimos métodos para convertir de binario a decimal (decimal) y de decimal a binario (binario).

El método ejecutar_instruccion se encarga de ejecutar la instrucción actual, interpretando el código binario y realizando la operación correspondiente.

Función main:

Creamos la función main, que es la entrada principal del programa.

Leemos las instrucciones desde un archivo de texto llamado "binario.txt" y las almacenamos en una lista bidimensional instrucc.

Creamos una instancia de la clase Instrucciones pasándole la lista de instrucciones como argumento.

Imprimimos el estado inicial del programa, mostrando el acumulador y las instrucciones.

Luego, en un bucle while, ejecutamos las instrucciones hasta que el valor del acumulador sea mayor o igual a 32.

CÓDIGO:

```
class Instrucciones:  #CLASE INSTRUCCIONES
    def __init__(self, instrucc):
        self.instrucc = instrucc
        self.acumulador = 1 #valor inicial del acumulador
        self.index = 0  #indice para la posicion actual

    def decimal(self, fila): #convertir de binario a decimal
        valor_decimal = 0
        for i in range(3, 8):
            potencia = 1
            for j in range(7 - i):
                potencia *= 2
            valor_decimal += self.instrucc[fila][i] * potencia
        return valor_decimal

    def binario(self, fila, acum): #convertir de decimal a binario y almacenarlo
        for i in range(7, 2, -1):
            self.instrucc[fila][i] = 0
        for i in range(7, 2, -1):
            if acum == 1:
                self.instrucc[fila][i] = 1
                break
            else:
                self.instrucc[fila][i] = acum % 2
                acum //= 2

    def ejecutar_instruccion(self):
        instruccion = self.instrucc[self.index][0] * 100 + self.instrucc[self.index][1] * 10 +
self.instrucc[self.index][2]
        direccion = 0

        if instruccion == 100:
            direccion = self.decimal(self.index)
            self.acumulador = self.decimal(direccion)

        elif instruccion == 101:
            direccion = self.decimal(self.index)
            self.acumulador += self.decimal(direccion)

        elif instruccion == 110:
            direccion = self.decimal(self.index)
            self.binario(direccion, self.acumulador)

        elif instruccion == 10:
            self.index = 0
            print("\n\n     ", self.acumulador)
            for k in range(8):
                print(k, end=". ")
                for j in range(8):
                    print(self.instrucc[k][j], end="")
                print()

        else:
            print("\nEsa instrucción no existe")

        self.index += 1
```

```python
def main():
    instrucc = [[0] * 8 for _ in range(8)]

    with open("binario.txt", "r") as f:
        i = 0
        for line in f:
            for j in range(8):
                instrucc[i][j] = int(line[j])
            i += 1

    programa = Instrucciones(instrucc)

    print("     ", programa.acumulador)
    for i in range(8):
        print(i, end=". ")
        for j in range(8):
            print(instrucc[i][j], end="")
        print()

    while programa.acumulador < 32:
        programa.ejecutar_instruccion()


if __name__ == "__main__":
    main()
```

PRUEBAS Y RESULTADOS:

```
PS C:\Users\sergi\OneDrive\Documentos\PARADIGMAS DE PROGRAMACION> & C:/Users/sergi/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/serg
i/OneDrive/Documentos/PARADIGMAS DE PROGRAMACION/ensamblador.py"
      1
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00000001
7. 00000001


      2
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00000010
7. 00000001


      3
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00000011
7. 00000001


      4
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
```

```
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00011100
7. 00000001


     29
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00011101
7. 00000001


     30
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00011110
7. 00000001


     31
0. 10000110
1. 10100111
2. 11000110
3. 01000001
4. 00000000
5. 00000000
6. 00011111
7. 00000001
PS C:\Users\sergi\OneDrive\Documentos\PARADIGMAS DE PROGRAMACION>
```