**Project construction content document CPU processing visualization program (Center Process Unit).**

1. **Idea:**

Dự án trực quan hóa hoạt động CPU là một ý tưởng rất hay và giáo dục, đặc biệt phù hợp cho môn Kiến trúc Máy tính và Hợp ngữ. Nó sẽ giúp bạn và những người khác hiểu rõ hơn về cách CPU xử lý các lệnh.

**Gợi ý chung về dự án này**

Đây là một dự án đầy thử thách nhưng rất đáng giá. Việc trực quan hóa các đường tín hiệu và cập nhật trạng thái theo thời gian thực sẽ là phần phức tạp nhất. Tuy nhiên, nếu thành công, nó sẽ tạo ra một công cụ học tập cực kỳ mạnh mẽ.

* **Tính chính xác là chìa khóa**: Bạn cần phải hiểu rất rõ về từng pha trong chu kỳ lệnh (Fetch, Decode, Execute) và cách các thành phần CPU (PC, IR, Registers, ALU, Control Unit, RAM) tương tác.
* **Thiết kế dữ liệu**: Cách bạn biểu diễn trạng thái của các thanh ghi, RAM, và các tín hiệu sẽ rất quan trọng.
* **Trạng thái thời gian thực**: Việc cập nhật và hiển thị trạng thái động (các đốm sáng, giá trị thanh ghi) đòi hỏi một vòng lặp sự kiện (event loop) được quản lý tốt.

**Tôi nên viết bằng Python hay các ngôn ngữ khác?**

**Python là một lựa chọn rất tốt cho dự án này**, đặc biệt nếu bạn đã quen thuộc với nó.

**Ưu điểm của Python:**

* **Phát triển nhanh**: Python có cú pháp rõ ràng, dễ đọc, giúp bạn triển khai ý tưởng nhanh chóng.
* **Thư viện đồ họa phong phú**: Tkinter (có sẵn), PyQt (mạnh mẽ hơn), Kivy... đều là các lựa chọn tốt để xây dựng GUI.
* **Dễ dàng xử lý logic**: Việc mô phỏng các phép toán bit, đọc/ghi RAM, quản lý trạng thái rất thuận tiện trong Python.
* **Tính giáo dục**: Mã nguồn Python sẽ dễ hiểu và dễ chia sẻ cho mục đích học tập hơn.

**Nhược điểm (nhỏ) của Python:**

* **Hiệu suất**: Với một số lượng lớn các "đốm sáng" chạy liên tục và cập nhật nhiều thành phần trên GUI, Python có thể không nhanh bằng C++ hoặc Java. Tuy nhiên, với quy mô một CPU đơn giản (16 địa chỉ RAM, vài thanh ghi), hiệu suất Python vẫn hoàn toàn chấp nhận được.

**Các ngôn ngữ khác (C++/Java):**

* **C++**: Sẽ cung cấp hiệu suất tốt nhất và kiểm soát bộ nhớ chặt chẽ. Nhưng thời gian phát triển GUI sẽ lâu hơn và phức tạp hơn (ví dụ với Qt/MFC).
* **Java**: Hiệu suất tốt, có Swing/JavaFX cho GUI. Cũng là một lựa chọn khả thi nhưng có thể ít "thú vị" hơn Python cho một dự án trực quan hóa nhanh.

**Lời khuyên:** Hãy tiếp tục với **Python**. Với yêu cầu trực quan hóa và mục đích giáo dục, Python mang lại sự cân bằng tốt nhất giữa tốc độ phát triển và khả năng thể hiện ý tưởng.

**Tôi nên dùng công cụ nào thì dễ và hiệu quả để xây dựng chương trình này?**

Bạn đã đề cập đến Tkinter và PyQt6. Cả hai đều là những lựa chọn hợp lý, nhưng có sự khác biệt:

1. **Tkinter**:
   * **Ưu điểm**: Có sẵn trong Python, không cần cài đặt thêm. Rất đơn giản để bắt đầu, lý tưởng cho những người mới làm quen với GUI.
   * **Nhược điểm**: Giao diện có thể không hiện đại và "lung linh" bằng PyQt. Các tính năng nâng cao hơn (ví dụ: vẽ đường cong phức tạp, hoạt ảnh mượt mà) có thể khó triển khai hơn.
   * **Phù hợp cho**: Dự án nhỏ, cần triển khai nhanh, không yêu cầu GUI quá cầu kỳ.
2. **PyQt6 (hoặc PyQt5)**:
   * **Ưu điểm**: Mạnh mẽ, linh hoạt, tạo ra giao diện chuyên nghiệp và hiện đại. Hỗ trợ nhiều tính năng đồ họa nâng cao (Custom Painting, QGraphicsScene/View cho đồ thị phức tạp, Animation Framework). Có Qt Designer giúp thiết kế GUI bằng cách kéo thả.
   * **Nhược điểm**: Cần cài đặt thư viện ngoài (pip install PyQt6). Có thể có đường cong học tập dốc hơn một chút so với Tkinter.
   * **Phù hợp cho**: Dự án có yêu cầu cao về mặt đồ họa, cần hoạt ảnh mượt mà, giao diện đẹp mắt, và sẵn sàng đầu tư thời gian học thêm.

**Lời khuyên về công cụ:** Với yêu cầu "có các đốm sáng chạy trên các đường dây đại diện cho tín hiệu điện", **PyQt6 (hoặc PyQt5)** sẽ là lựa chọn hiệu quả và dễ dàng hơn để triển khai các hoạt ảnh động và vẽ đường dây tùy chỉnh. QGraphicsScene và QGraphicsView của PyQt rất mạnh mẽ cho việc này.

**Tôi nên triển khai theo hướng Web hay App hay chương trình sẽ hợp lý hơn?**

**Triển khai dưới dạng Ứng dụng Desktop (Desktop App) là hợp lý nhất** cho dự án này.

**Tại sao?**

* **Kiểm soát tài nguyên và hiệu suất**: Ứng dụng Desktop có quyền truy cập trực tiếp hơn vào tài nguyên hệ thống, giúp kiểm soát hiệu suất tốt hơn cho các hoạt ảnh thời gian thực.
* **Độ phức tạp**: Xây dựng một ứng dụng web với các yêu cầu tương tự (trực quan hóa đường tín hiệu động, cập nhật trạng thái liên tục) sẽ phức tạp hơn nhiều. Bạn sẽ cần JavaScript, HTML Canvas/SVG, và một kiến thức vững chắc về lập trình Frontend.
* **Mục đích học tập**: Nếu mục đích chính là học về Kiến trúc Máy tính và Hợp ngữ, việc phát triển một ứng dụng Desktop sẽ giúp bạn tập trung vào logic CPU mà không bị phân tâm quá nhiều bởi các công nghệ Web phức tạp.
* **Môi trường trình bày**: Một ứng dụng desktop độc lập thường dễ dàng trình bày trong môi trường học thuật hơn.

**Kết luận:**

* **Ngôn ngữ**: **Python**.
* **Công cụ GUI**: **PyQt6** (hoặc PyQt5) để có đồ họa tốt nhất và khả năng hoạt ảnh.
* **Hướng triển khai**: **Ứng dụng Desktop**.

1. **Deploy Project with Python PyQt 6:**

Chúng ta sẽ đi sâu vào hướng dẫn chi tiết cách triển khai dự án trực quan hóa CPU bằng Python và thư viện PyQt6. Dự án này sẽ có nhiều phần, nên tôi sẽ chia thành các giai đoạn để bạn dễ hình dung và thực hiện.

### Giai đoạn 1: Chuẩn bị Dự án & Giao diện Cơ bản

Đây là bước nền tảng để tạo cửa sổ ứng dụng và hiển thị sơ đồ CPU tĩnh.

#### 1. Cấu trúc Dự án (Project Structure)

Để dự án được gọn gàng và dễ quản lý, bạn nên tổ chức các file như sau:

cpu\_visualizer/

├── src/

│ ├── main.py # Điểm khởi chạy chính của ứng dụng

│ ├── cpu\_core.py # Chứa logic mô phỏng CPU (registers, RAM, ALU, CU)

│ ├── gui\_elements.py # Chứa các lớp/hàm liên quan đến phần GUI động (đốm sáng, cập nhật giá trị)

│ └── \_\_init\_\_.py # Đánh dấu thư mục src là một gói Python

├── assets/

│ ├── cpu\_diagram.jpg # File ảnh sơ đồ CPU (hình bạn cung cấp)

│ └── icons/ # Thư mục chứa các icon nếu có

├── README.md # Tài liệu dự án

└── requirements.txt # Danh sách các thư viện cần cài đặt

#### 2. Cài đặt PyQt6

Đảm bảo bạn đã cài đặt PyQt6. Mở Terminal (hoặc Command Prompt) và chạy lệnh sau:

Bash

pip install PyQt6

#### 3. Tạo Cửa sổ Chính (main.py)

Hãy bắt đầu bằng cách tạo một cửa sổ PyQt6 cơ bản và hiển thị sơ đồ CPU làm nền.

**File: src/main.py**

Python

import sys

from PyQt6.QtWidgets import QApplication, QMainWindow, QWidget, QVBoxLayout, QLabel, QPushButton, QTableWidget, QTableWidgetItem, QGraphicsView, QGraphicsScene

from PyQt6.QtGui import QPixmap, QImage, QColor, QPen

from PyQt6.QtCore import Qt, QTimer, QPointF

# Import logic CPU sau này sẽ được thêm vào

# from .cpu\_core import CPU # Sẽ được thêm vào sau

class CPUVisualizerApp(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("CPU Execution Visualizer")

self.setGeometry(100, 100, 1200, 700) # Kích thước cửa sổ ban đầu

self.central\_widget = QWidget()

self.setCentralWidget(self.central\_widget)

self.main\_layout = QVBoxLayout(self.central\_widget)

self.init\_ui()

def init\_ui(self):

# 1. Load và hiển thị sơ đồ CPU làm nền

self.scene = QGraphicsScene()

self.view = QGraphicsView(self.scene)

self.main\_layout.addWidget(self.view)

# Tải ảnh sơ đồ CPU (đảm bảo đường dẫn đúng)

try:

self.cpu\_diagram\_pixmap = QPixmap("assets/image\_212fe2.jpg")

if self.cpu\_diagram\_pixmap.isNull():

print("Lỗi: Không thể tải ảnh sơ đồ CPU. Kiểm tra đường dẫn.")

# Nếu ảnh không tải được, bạn có thể tạo một hình chữ nhật màu để thay thế

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray"))

else:

self.scene.addPixmap(self.cpu\_diagram\_pixmap)

# Đặt kích thước scene bằng kích thước ảnh

self.scene.setSceneRect(self.cpu\_diagram\_pixmap.rect())

self.view.fitInView(self.scene.sceneRect(), Qt.AspectRatioMode.KeepAspectRatio) # Đảm bảo ảnh vừa khung nhìn

except Exception as e:

print(f"Lỗi khi tải ảnh: {e}")

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray")) # Fallback

# Thêm các thành phần UI khác (RAM, Registers, Controls)

self.setup\_ram\_table()

self.setup\_control\_panel()

# ... (các phần khác sẽ được thêm vào sau)

def setup\_ram\_table(self):

# Đây là ví dụ về cách thêm bảng RAM

self.ram\_table = QTableWidget(16, 2) # 16 địa chỉ, 2 cột (Address, Data)

self.ram\_table.setHorizontalHeaderLabels(["Address", "Data (8-bit)"])

# Điền địa chỉ RAM

for i in range(16):

self.ram\_table.setItem(i, 0, QTableWidgetItem(str(i)))

self.ram\_table.setItem(i, 1, QTableWidgetItem("00000000")) # Giá trị mặc định

# Cài đặt kích thước cột

self.ram\_table.setColumnWidth(0, 70)

self.ram\_table.setColumnWidth(1, 100)

# Có thể thêm vào một layout khác hoặc một dock widget

# self.main\_layout.addWidget(self.ram\_table) # Tạm thời thêm vào đây để kiểm tra

def setup\_control\_panel(self):

# Ví dụ về các nút điều khiển

self.run\_button = QPushButton("Run")

self.step\_button = QPushButton("Step")

self.reset\_button = QPushButton("Reset")

# Thêm vào một layout hoặc dock widget

# self.main\_layout.addWidget(self.run\_button) # Tạm thời

if \_\_name\_\_ == "\_\_main\_\_":

app = QApplication(sys.argv)

window = CPUVisualizerApp()

window.show()

sys.exit(app.exec())

**Cách chạy bước này:**

1. Lưu đoạn code trên vào src/main.py.
2. Đặt ảnh image\_212fe2.jpg vào thư mục assets/.
3. Mở terminal tại thư mục gốc của dự án (cpu\_visualizer/).
4. Chạy lệnh: python src/main.py Bạn sẽ thấy một cửa sổ với sơ đồ CPU làm nền.

### Giai đoạn 2: Mô hình Dữ liệu - Biểu diễn trạng thái CPU và Lệnh

Đây là trái tim của dự án, nơi bạn sẽ định nghĩa cách dữ liệu (RAM, Registers, Instructions) được lưu trữ và tương tác. Bạn sẽ làm việc trong file src/cpu\_core.py.

#### 1. Mô hình RAM (Memory Model)

RAM của bạn có 16 địa chỉ, mỗi địa chỉ chứa 8 bit dữ liệu.

**File: src/cpu\_core.py**

Python

# cpu\_core.py

class CPU:

def \_\_init\_\_(self):

# Khởi tạo RAM: 16 địa chỉ, mỗi địa chỉ 8 bit (sử dụng list chứa chuỗi bit hoặc int)

# Tùy chọn 1: list các chuỗi bit (dễ trực quan hóa)

self.ram = ["00000000"] \* 16 # 16 địa chỉ, mỗi địa chỉ 8 bit, khởi tạo là 0

# Tùy chọn 2: list các số nguyên (dễ tính toán hơn)

# self.ram = [0] \* 16 # 16 địa chỉ, mỗi địa chỉ 8 bit (0-255)

# Khởi tạo các thanh ghi (Registers)

self.registers = {

'A': 0, # Register A

'B': 0, # Register B

'C': 0, # Register C

'D': 0, # Register D

'IR\_Opcode': '0000', # Instruction Register - Opcode part (4 bits)

'IR\_AddrData': '0000', # Instruction Register - Address/Data part (4 bits)

'IAR': 0, # Instruction Address Register (Program Counter - PC) - 4 bits (0-15)

}

# Khởi tạo các cờ trạng thái (Flags)

self.flags = {

'O': False, # Overflow Flag

'Z': False, # Zero Flag

'N': False # Negative Flag

}

# Bảng mã lệnh (Opcode definitions)

# Đây là nơi bạn định nghĩa các mã Opcode và ý nghĩa của chúng

self.opcode\_map = {

'0000': {'name': 'NOP', 'description': 'No Operation'},

'0001': {'name': 'LOAD\_A', 'description': 'Load value from RAM[Addr] to Register A', 'operands': 'Addr'},

'0010': {'name': 'LOAD\_B', 'description': 'Load value from RAM[Addr] to Register B', 'operands': 'Addr'},

'0011': {'name': 'STORE\_A', 'description': 'Store value from Register A to RAM[Addr]', 'operands': 'Addr'},

'0100': {'name': 'STORE\_B', 'description': 'Store value from Register B to RAM[Addr]', 'operands': 'Addr'},

'0101': {'name': 'ADD', 'description': 'A = A + B', 'operands': 'None'},

'0110': {'name': 'SUB', 'description': 'A = A - B', 'operands': 'None'},

'0111': {'name': 'JUMP', 'description': 'Jump to instruction at RAM[Addr]', 'operands': 'Addr'},

'1000': {'name': 'JUMP\_NEG', 'description': 'Jump to RAM[Addr] if N flag is set', 'operands': 'Addr'},

'1001': {'name': 'JUMP\_ZERO', 'description': 'Jump to RAM[Addr] if Z flag is set', 'operands': 'Addr'},

# Thêm các lệnh khác tùy theo thiết kế CPU của bạn

'1111': {'name': 'HALT', 'description': 'Stop program execution', 'operands': 'None'}

}

def read\_ram(self, address):

"""Đọc giá trị 8-bit từ địa chỉ RAM."""

if 0 <= address < len(self.ram):

# Nếu RAM lưu chuỗi bit:

return self.ram[address]

# Nếu RAM lưu số nguyên:

# return f"{self.ram[address]:08b}" # Chuyển số nguyên thành chuỗi bit 8 ký tự

else:

raise ValueError(f"Địa chỉ RAM không hợp lệ: {address}")

def write\_ram(self, address, data):

"""Ghi giá trị 8-bit vào địa chỉ RAM."""

if 0 <= address < len(self.ram):

# Nếu RAM lưu chuỗi bit:

if not isinstance(data, str) or len(data) != 8 or not all(c in '01' for c in data):

raise ValueError("Dữ liệu RAM phải là chuỗi 8 bit.")

self.ram[address] = data

# Nếu RAM lưu số nguyên:

# self.ram[address] = int(data, 2) # Chuyển chuỗi bit thành số nguyên

else:

raise ValueError(f"Địa chỉ RAM không hợp lệ: {address}")

def load\_instruction\_to\_ram(self, address, opcode\_str, operand\_str):

"""Nạp một lệnh 8-bit vào RAM (4 bit opcode + 4 bit operand/address)."""

if len(opcode\_str) != 4 or not all(c in '01' for c in opcode\_str):

raise ValueError("Opcode phải là chuỗi 4 bit.")

if len(operand\_str) != 4 or not all(c in '01' for c in operand\_str):

raise ValueError("Operand phải là chuỗi 4 bit.")

full\_instruction = opcode\_str + operand\_str

self.write\_ram(address, full\_instruction)

print(f"Nạp lệnh: RAM[{address}] = {full\_instruction} ({self.opcode\_map.get(opcode\_str, {}).get('name', 'UNKNOWN')})")

def reset(self):

"""Đặt lại trạng thái CPU về ban đầu."""

self.ram = ["00000000"] \* 16

self.registers = {

'A': 0, 'B': 0, 'C': 0, 'D': 0,

'IR\_Opcode': '0000', 'IR\_AddrData': '0000',

'IAR': 0, # Program Counter

}

self.flags = {'O': False, 'Z': False, 'N': False}

print("CPU đã được reset.")

#### 2. Định nghĩa tập lệnh (Instruction Set Definition)

Trong file src/cpu\_core.py ở trên, tôi đã thêm self.opcode\_map. Đây là một từ điển định nghĩa các lệnh của CPU của bạn.

* Mỗi khóa là một chuỗi 4 bit (opcode).
* Giá trị là một từ điển chứa:
  + 'name': Tên dễ đọc của lệnh.
  + 'description': Mô tả chức năng.
  + 'operands': Kiểu operand (ví dụ: 'Addr' nếu lệnh cần một địa chỉ RAM, 'None' nếu không cần).

#### 3. Các hàm cơ bản để tương tác với RAM và Registers

* read\_ram(self, address): Đọc giá trị 8-bit từ một địa chỉ RAM cụ thể.
* write\_ram(self, address, data): Ghi giá trị 8-bit vào một địa chỉ RAM.
* load\_instruction\_to\_ram(self, address, opcode\_str, operand\_str): Hàm tiện ích để nạp một lệnh 8-bit (kết hợp opcode và operand) vào RAM.

### Các bước tiếp theo:

1. **Cập nhật main.py để kết nối với cpu\_core.py**: Bạn cần tạo một instance của lớp CPU trong CPUVisualizerApp và sử dụng nó để quản lý trạng thái.
2. **Hiển thị RAM trên GUI**: Liên kết self.ram\_table với dữ liệu self.cpu.ram để hiển thị và cập nhật.
3. **Thêm logic nạp lệnh**: Tạo các trường nhập liệu hoặc danh sách chọn lệnh để người dùng có thể nhập các lệnh vào RAM.

1. **CPU Diagram**
   1. **CPU Diagram**

|  |  |
| --- | --- |
| **Address** | **Data** |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |
| **5** |  |
| **6** |  |
| **7** |  |
| **8** |  |
| **9** |  |
| **10** |  |
| **11** |  |
| **12** |  |
| **13** |  |
| **14** |  |
| **15** |  |
| **16** |  |

**RAM**

**CONTROL**

**UNIT**

**INSTRUCTION RGISTER**

**O**

**Z**

**N**

**INST.ADDR.REGISTER**

**OPCODE**

**INPUT**

**INPUT**

**OUTPUT**

**FLATS**

**OPERATION CODE**

**DATA**

**REGISTER C**

**REGISTER D**

**REGISTER B**

**REGISTER A**

**ADDRESS INPUT**

**OPCODE**

**READ ENABLE**

**WRITE ENABLE**

**... ...**

1. **Xây dựng CPU\_Core và Liên Kết với Main**

Bây giờ chúng ta sẽ tích hợp lớp CPU vào ứng dụng chính và bắt đầu xây dựng các chức năng cốt lõi của chu trình CPU: Fetch (Nạp lệnh), Decode (Giải mã lệnh), và Execute (Thực thi lệnh).

### Giai đoạn 3: Logic Mô phỏng CPU & Tích hợp vào GUI

Chúng ta sẽ bổ sung các phương thức vào lớp CPU trong cpu\_core.py và sau đó kết nối chúng với giao diện người dùng trong main.py.

#### 1. Cập nhật file src/cpu\_core.py

Chúng ta sẽ thêm các phương thức fetch\_instruction, decode\_instruction, execute\_instruction, update\_flags và step\_cycle vào lớp CPU.

**File: src/cpu\_core.py (Cập nhật)**

Python

# cpu\_core.py

class CPU:

def \_\_init\_\_(self):

# Khởi tạo RAM: 16 địa chỉ, mỗi địa chỉ 8 bit (sử dụng list chứa chuỗi bit)

self.ram = ["00000000"] \* 16 # 16 địa chỉ, mỗi địa chỉ 8 bit, khởi tạo là 0

# Khởi tạo các thanh ghi (Registers)

self.registers = {

'A': 0, # Register A (8-bit value, 0-255)

'B': 0, # Register B (8-bit value, 0-255)

# Các thanh ghi C, D có thể thêm vào nếu cần cho các lệnh khác

# 'C': 0,

# 'D': 0,

'IR\_Opcode': '0000', # Instruction Register - Opcode part (4 bits string)

'IR\_AddrData': '0000', # Instruction Register - Address/Data part (4 bits string)

'IAR': 0, # Instruction Address Register (Program Counter - PC) - 4 bits (0-15)

}

# Khởi tạo các cờ trạng thái (Flags)

self.flags = {

'O': False, # Overflow Flag

'Z': False, # Zero Flag

'N': False # Negative Flag

}

# Bảng mã lệnh (Opcode definitions)

self.opcode\_map = {

'0000': {'name': 'NOP', 'description': 'No Operation', 'operands': 'None'},

'0001': {'name': 'LOAD\_A', 'description': 'Load value from RAM[Addr] into Reg A', 'operands': 'Addr'},

'0010': {'name': 'LOAD\_B', 'description': 'Load value from RAM[Addr] into Reg B', 'operands': 'Addr'},

'0011': {'name': 'STORE\_A', 'description': 'Store value from Reg A to RAM[Addr]', 'operands': 'Addr'},

'0100': {'name': 'STORE\_B', 'description': 'Store value from Reg B to RAM[Addr]', 'operands': 'Addr'},

'0101': {'name': 'ADD', 'description': 'Reg A = Reg A + Reg B', 'operands': 'None'},

'0110': {'name': 'SUB', 'description': 'Reg A = Reg A - Reg B', 'operands': 'None'},

'0111': {'name': 'JUMP', 'description': 'Jump to instruction at RAM[Addr]', 'operands': 'Addr'},

'1000': {'name': 'JUMP\_NEG', 'description': 'Jump to RAM[Addr] if N flag is set', 'operands': 'Addr'},

'1001': {'name': 'JUMP\_ZERO', 'description': 'Jump to RAM[Addr] if Z flag is set', 'operands': 'Addr'},

'1111': {'name': 'HALT', 'description': 'Stop program execution', 'operands': 'None'}

}

self.is\_halted = False # Cờ báo hiệu CPU đã dừng

def read\_ram(self, address):

"""Đọc giá trị 8-bit từ địa chỉ RAM."""

if 0 <= address < len(self.ram):

return self.ram[address] # Trả về chuỗi bit 8 ký tự

else:

print(f"Lỗi: Địa chỉ RAM không hợp lệ khi đọc: {address}")

return "00000000" # Trả về giá trị mặc định khi lỗi

def write\_ram(self, address, data\_str):

"""Ghi giá trị 8-bit vào địa chỉ RAM."""

if not isinstance(data\_str, str) or len(data\_str) != 8 or not all(c in '01' for c in data\_str):

raise ValueError(f"Dữ liệu RAM phải là chuỗi 8 bit. Nhận được: {data\_str}")

if 0 <= address < len(self.ram):

self.ram[address] = data\_str

else:

print(f"Lỗi: Địa chỉ RAM không hợp lệ khi ghi: {address}")

def load\_instruction\_to\_ram(self, address, full\_instruction\_str):

"""

Nạp một lệnh 8-bit (chuỗi nhị phân) vào RAM.

Kiểm tra độ dài và định dạng.

"""

if not isinstance(full\_instruction\_str, str) or len(full\_instruction\_str) != 8 or not all(c in '01' for c in full\_instruction\_str):

raise ValueError(f"Lệnh phải là chuỗi 8 bit nhị phân. Nhận được: {full\_instruction\_str}")

self.write\_ram(address, full\_instruction\_str)

opcode\_str = full\_instruction\_str[:4]

operand\_str = full\_instruction\_str[4:]

op\_info = self.opcode\_map.get(opcode\_str, {'name': 'UNKNOWN'})

print(f"Nạp lệnh: RAM[{address}] = {full\_instruction\_str} ({op\_info['name']} {int(operand\_str, 2) if op\_info.get('operands') == 'Addr' else ''})")

def \_update\_flags(self, value):

"""Cập nhật các cờ trạng thái (Z, N, O) dựa trên giá trị kết quả."""

# Giả sử giá trị là số nguyên 8-bit signed (-128 đến 127) hoặc unsigned (0-255)

# Để đơn giản, ta sẽ dùng 8-bit unsigned (0-255) và kiểm tra tràn số cho cộng/trừ.

# Zero Flag

self.flags['Z'] = (value == 0)

# Negative Flag (bit cao nhất của 8-bit là 1)

# Giả định: nếu giá trị > 127 (0b01111111), thì coi là âm trong biểu diễn signed 8-bit

# Hoặc đơn giản hơn: nếu kết quả phép trừ là âm.

self.flags['N'] = (value < 0) # Đối với kết quả tạm thời, có thể là âm

# Overflow Flag: Cần xử lý trong ADD/SUB cụ thể

# Để đơn giản hóa, ta sẽ đặt giới hạn 0-255 cho Reg A, B.

# Nếu kết quả vượt quá 255 hoặc nhỏ hơn 0, thì cờ O được bật.

# Ở đây chỉ là placeholder, logic cụ thể sẽ nằm trong execute\_instruction

self.flags['O'] = False # Reset mặc định, cập nhật trong phép toán

def fetch\_instruction(self):

"""

Pha Fetch: Nạp lệnh từ RAM vào thanh ghi lệnh (IR).

Cập nhật IAR (Program Counter).

"""

if self.is\_halted:

return None # Không fetch nếu CPU đã dừng

current\_address = self.registers['IAR']

instruction\_bits = self.read\_ram(current\_address) # Lấy chuỗi 8-bit lệnh

self.registers['IR\_Opcode'] = instruction\_bits[:4]

self.registers['IR\_AddrData'] = instruction\_bits[4:]

print(f"Fetch: RAM[{current\_address}] -> IR ({self.registers['IR\_Opcode']} {self.registers['IR\_AddrData']})")

# Tăng IAR lên cho lệnh tiếp theo (nếu không có JUMP)

self.registers['IAR'] = (self.registers['IAR'] + 1) % len(self.ram) # Đảm bảo IAR không vượt quá kích thước RAM

return instruction\_bits

def decode\_instruction(self):

"""

Pha Decode: Giải mã lệnh từ IR để xác định Opcode và Operand.

"""

opcode\_str = self.registers['IR\_Opcode']

operand\_str = self.registers['IR\_AddrData']

instruction\_info = self.opcode\_map.get(opcode\_str)

if not instruction\_info:

print(f"Decode: Lệnh không xác định: {opcode\_str}")

return {'name': 'UNKNOWN', 'operands': 'None', 'opcode\_str': opcode\_str, 'operand\_val': None}

operand\_value = None

if instruction\_info['operands'] == 'Addr':

operand\_value = int(operand\_str, 2) # Chuyển chuỗi bit địa chỉ thành số nguyên

# Nếu 'operands' là 'Data' thì cũng có thể dùng int(operand\_str, 2)

print(f"Decode: Opcode: {instruction\_info['name']} (Operand: {operand\_value if operand\_value is not None else 'N/A'})")

return {'name': instruction\_info['name'],

'operands': instruction\_info['operands'],

'opcode\_str': opcode\_str,

'operand\_val': operand\_value}

def execute\_instruction(self, decoded\_instruction):

"""

Pha Execute: Thực thi lệnh đã giải mã.

"""

op\_name = decoded\_instruction['name']

operand\_val = decoded\_instruction['operand\_val']

print(f"Execute: {op\_name}")

if op\_name == 'LOAD\_A':

data\_from\_ram = self.read\_ram(operand\_val)

self.registers['A'] = int(data\_from\_ram, 2) # Chuyển chuỗi bit sang int

print(f" Reg A = RAM[{operand\_val}] = {self.registers['A']}")

self.\_update\_flags(self.registers['A']) # Cập nhật cờ sau khi nạp

elif op\_name == 'LOAD\_B':

data\_from\_ram = self.read\_ram(operand\_val)

self.registers['B'] = int(data\_from\_ram, 2)

print(f" Reg B = RAM[{operand\_val}] = {self.registers['B']}")

self.\_update\_flags(self.registers['B']) # Cập nhật cờ sau khi nạp

elif op\_name == 'STORE\_A':

data\_to\_store = f"{self.registers['A']:08b}" # Chuyển int sang chuỗi bit 8 ký tự

self.write\_ram(operand\_val, data\_to\_store)

print(f" RAM[{operand\_val}] = Reg A ({self.registers['A']})")

elif op\_name == 'STORE\_B':

data\_to\_store = f"{self.registers['B']:08b}"

self.write\_ram(operand\_val, data\_to\_store)

print(f" RAM[{operand\_val}] = Reg B ({self.registers['B']})")

elif op\_name == 'ADD':

result = self.registers['A'] + self.registers['B']

# Kiểm tra tràn số (8-bit unsigned: 0-255)

self.flags['O'] = (result > 255)

self.registers['A'] = result & 0xFF # Giới hạn kết quả trong 8 bit (0-255)

print(f" Reg A = {self.registers['A']} + {self.registers['B']} = {self.registers['A']}")

self.\_update\_flags(self.registers['A']) # Cập nhật cờ sau phép toán

elif op\_name == 'SUB':

result = self.registers['A'] - self.registers['B']

# Kiểm tra tràn số (8-bit unsigned: 0-255) hoặc âm

self.flags['O'] = (result < 0) # Coi là tràn nếu kết quả âm (trong unsigned)

self.registers['A'] = result & 0xFF # Giới hạn kết quả trong 8 bit (0-255)

print(f" Reg A = {self.registers['A']} - {self.registers['B']} = {self.registers['A']}")

self.\_update\_flags(self.registers['A']) # Cập nhật cờ sau phép toán

elif op\_name == 'JUMP':

self.registers['IAR'] = operand\_val # Ghi đè IAR bằng địa chỉ đích

print(f" JUMP to address {operand\_val}")

elif op\_name == 'JUMP\_NEG':

if self.flags['N']:

self.registers['IAR'] = operand\_val

print(f" N flag set, JUMP to address {operand\_val}")

else:

print(" N flag not set, JUMP\_NEG skipped.")

elif op\_name == 'JUMP\_ZERO':

if self.flags['Z']:

self.registers['IAR'] = operand\_val

print(f" Z flag set, JUMP to address {operand\_val}")

else:

print(" Z flag not set, JUMP\_ZERO skipped.")

elif op\_name == 'HALT':

self.is\_halted = True

print("CPU Halted.")

elif op\_name == 'UNKNOWN':

print(" Lỗi: Lệnh không xác định. Dừng thực thi.")

self.is\_halted = True

# Sau mỗi lệnh, cập nhật trạng thái CPU lên GUI (sẽ làm sau)

# self.update\_gui\_status()

def step\_cycle(self):

"""Thực hiện một chu kỳ Fetch-Decode-Execute."""

if self.is\_halted:

print("CPU đã dừng. Vui lòng Reset.")

return False # Không thực hiện bước nào nữa

print(f"\n--- Chu kỳ CPU (IAR: {self.registers['IAR']}) ---")

instruction\_bits = self.fetch\_instruction()

if instruction\_bits is None: # CPU halted during fetch

return False

decoded\_instruction = self.decode\_instruction()

self.execute\_instruction(decoded\_instruction)

# Kiểm tra xem có lệnh HALT không

if self.is\_halted:

return False # Báo hiệu đã dừng

return True # Báo hiệu đã thực hiện xong một bước

def reset(self):

"""Đặt lại trạng thái CPU về ban đầu."""

super().\_\_init\_\_() # Call parent reset if any, though not explicitly defined here

self.ram = ["00000000"] \* 16

self.registers = {

'A': 0, 'B': 0, # 'C': 0, 'D': 0,

'IR\_Opcode': '0000', 'IR\_AddrData': '0000',

'IAR': 0,

}

self.flags = {'O': False, 'Z': False, 'N': False}

self.is\_halted = False

print("CPU đã được reset.")

**Lưu ý quan trọng trong cpu\_core.py:**

* **Định dạng dữ liệu**: Tôi đã chọn lưu trữ dữ liệu RAM và các phần của IR dưới dạng chuỗi bit ("00000000"). Điều này giúp dễ dàng liên kết với biểu diễn nhị phân trên sơ đồ. Khi cần tính toán (như trong ADD, SUB), chúng ta chuyển đổi chuỗi bit sang số nguyên bằng int(bit\_string, 2). Khi ghi lại, chuyển lại thành chuỗi bit bằng f"{int\_value:08b}".
* **Flag O, Z, N**:
  + Z (Zero Flag) được đặt nếu kết quả phép toán bằng 0.
  + N (Negative Flag) được đặt nếu kết quả phép toán là âm. (Lưu ý: trong CPU thực, N-flag thường là bit dấu của kết quả, nhưng ở đây ta đơn giản hóa).
  + O (Overflow Flag) được đặt nếu kết quả vượt quá giới hạn 8-bit unsigned (0-255).
* **step\_cycle()**: Hàm này sẽ là trái tim của việc mô phỏng từng bước. Nó gọi lần lượt fetch, decode, execute.
* **HALT instruction**: Khi lệnh HALT được thực thi, cờ self.is\_halted được bật, ngăn CPU thực hiện thêm các bước.
* **Input/Output trong cpu\_core.py**: Hiện tại, các hàm in ra console (print()) để bạn có thể theo dõi quá trình. Sau này, chúng ta sẽ thay thế hoặc bổ sung bằng việc cập nhật GUI.

#### 2. Cập nhật file src/main.py để tích hợp CPU và chức năng cốt lõi

Bây giờ, chúng ta sẽ tạo một instance của CPU trong ứng dụng chính và liên kết các nút Run, Step, Reset với các chức năng của CPU.

**File: src/main.py (Cập nhật)**

Python

import sys

from PyQt6.QtWidgets import (

QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,

QLabel, QPushButton, QTableWidget, QTableWidgetItem, QGraphicsView,

QGraphicsScene, QComboBox, QLineEdit, QFormLayout, QMessageBox

)

from PyQt6.QtGui import QPixmap, QColor, QPen

from PyQt6.QtCore import Qt, QTimer, QPointF

from .cpu\_core import CPU # Import lớp CPU từ file cpu\_core.py

class CPUVisualizerApp(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("CPU Execution Visualizer")

self.setGeometry(100, 100, 1200, 750) # Kích thước cửa sổ ban đầu

self.cpu = CPU() # Khởi tạo đối tượng CPU

self.central\_widget = QWidget()

self.setCentralWidget(self.central\_widget)

self.main\_layout = QHBoxLayout(self.central\_widget) # Dùng QHBoxLayout chính

self.init\_ui()

self.update\_gui\_cpu\_status() # Cập nhật trạng thái GUI lần đầu

def init\_ui(self):

# Left Panel (Controls, RAM Input, Registers, Flags)

self.left\_panel = QVBoxLayout()

self.main\_layout.addLayout(self.left\_panel, 1) # Chiếm 1 phần (ví dụ 1/3)

# Right Panel (CPU Diagram)

self.right\_panel = QVBoxLayout()

self.main\_layout.addLayout(self.right\_panel, 2) # Chiếm 2 phần (ví dụ 2/3)

# --- Right Panel: CPU Diagram ---

self.scene = QGraphicsScene()

self.view = QGraphicsView(self.scene)

self.right\_panel.addWidget(self.view)

try:

self.cpu\_diagram\_pixmap = QPixmap("assets/image\_212fe2.jpg")

if self.cpu\_diagram\_pixmap.isNull():

QMessageBox.critical(self, "Lỗi tải ảnh", "Không thể tải ảnh sơ đồ CPU. Kiểm tra đường dẫn 'assets/image\_212fe2.jpg'.")

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray"))

else:

self.scene.addPixmap(self.cpu\_diagram\_pixmap)

self.scene.setSceneRect(self.cpu\_diagram\_pixmap.rect())

self.view.fitInView(self.scene.sceneRect(), Qt.AspectRatioMode.KeepAspectRatio)

except Exception as e:

QMessageBox.critical(self, "Lỗi tải ảnh", f"Xảy ra lỗi khi tải ảnh: {e}")

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray")) # Fallback

# --- Left Panel: Controls and Data ---

# 1. Control Buttons (Run, Step, Reset)

self.control\_buttons\_layout = QHBoxLayout()

self.run\_button = QPushButton("Run")

self.step\_button = QPushButton("Step")

self.reset\_button = QPushButton("Reset")

self.control\_buttons\_layout.addWidget(self.run\_button)

self.control\_buttons\_layout.addWidget(self.step\_button)

self.control\_buttons\_layout.addWidget(self.reset\_button)

self.left\_panel.addLayout(self.control\_buttons\_layout)

# Connect buttons to CPU methods

self.run\_button.clicked.connect(self.run\_cpu)

self.step\_button.clicked.connect(self.step\_cpu)

self.reset\_button.clicked.connect(self.reset\_cpu)

# 2. RAM Table (for display and loading instructions)

self.ram\_group\_box = QWidget() # GroupBox for better organization

self.ram\_group\_box\_layout = QVBoxLayout(self.ram\_group\_box)

self.left\_panel.addWidget(self.ram\_group\_box)

self.ram\_table = QTableWidget(16, 2) # 16 địa chỉ, 2 cột (Address, Data)

self.ram\_table.setHorizontalHeaderLabels(["Address", "Data (8-bit)"])

for i in range(16):

self.ram\_table.setItem(i, 0, QTableWidgetItem(str(i)))

self.ram\_table.setItem(i, 1, QTableWidgetItem("00000000")) # Giá trị mặc định

self.ram\_table.setColumnWidth(0, 50)

self.ram\_table.setColumnWidth(1, 90)

self.ram\_table.verticalHeader().setVisible(False) # Hide row numbers

self.ram\_group\_box\_layout.addWidget(QLabel("RAM (Instruction/Data)"))

self.ram\_group\_box\_layout.addWidget(self.ram\_table)

# 3. Instruction Loader (Nạp lệnh vào RAM)

self.instr\_loader\_layout = QFormLayout()

self.left\_panel.addLayout(self.instr\_loader\_layout)

self.instr\_address\_input = QLineEdit("0")

self.instr\_address\_input.setPlaceholderText("0-15")

self.instr\_address\_input.setFixedWidth(50)

self.opcode\_combo = QComboBox()

# Thêm các opcode từ CPU.opcode\_map

for opcode\_bits, info in self.cpu.opcode\_map.items():

self.opcode\_combo.addItem(f"{info['name']} ({opcode\_bits})", opcode\_bits)

self.operand\_input = QLineEdit("0000")

self.operand\_input.setPlaceholderText("4-bit binary (e.g., 0101)")

self.operand\_input.setFixedWidth(80)

self.load\_instr\_button = QPushButton("Load Instruction to RAM")

self.load\_instr\_button.clicked.connect(self.load\_instruction\_gui)

self.instr\_loader\_layout.addRow(QLabel("Địa chỉ RAM:"), self.instr\_address\_input)

self.instr\_loader\_layout.addRow(QLabel("Opcode:"), self.opcode\_combo)

self.instr\_loader\_layout.addRow(QLabel("Operand/Addr:"), self.operand\_input)

self.instr\_loader\_layout.addRow(self.load\_instr\_button)

# 4. Register & Flag Display (Sẽ được chi tiết hóa sau)

self.register\_display\_labels = {}

self.flags\_display\_labels = {}

self.setup\_register\_and\_flag\_display() # Gọi hàm thiết lập ban đầu

# Timer cho chế độ Run (tự động chạy từng bước)

self.run\_timer = QTimer(self)

self.run\_timer.setInterval(500) # Mặc định 500ms (0.5 giây) mỗi bước

self.run\_timer.timeout.connect(self.step\_cpu)

def setup\_register\_and\_flag\_display(self):

# Đây là nơi bạn sẽ tạo các QLabel để hiển thị giá trị thanh ghi và cờ

# Ví dụ:

reg\_flag\_layout = QVBoxLayout()

self.left\_panel.addLayout(reg\_flag\_layout)

reg\_flag\_layout.addWidget(QLabel("--- CPU Status ---"))

# Registers

reg\_layout = QFormLayout()

self.register\_display\_labels['A'] = QLabel("Reg A: 0")

self.register\_display\_labels['B'] = QLabel("Reg B: 0")

self.register\_display\_labels['IAR'] = QLabel("IAR: 0")

self.register\_display\_labels['IR\_Opcode'] = QLabel("IR (Opcode): 0000")

self.register\_display\_labels['IR\_AddrData'] = QLabel("IR (Addr/Data): 0000")

reg\_layout.addRow(self.register\_display\_labels['A'])

reg\_layout.addRow(self.register\_display\_labels['B'])

reg\_layout.addRow(self.register\_display\_labels['IAR'])

reg\_layout.addRow(self.register\_display\_labels['IR\_Opcode'])

reg\_layout.addRow(self.register\_display\_labels['IR\_AddrData'])

reg\_flag\_layout.addLayout(reg\_layout)

# Flags

flag\_layout = QHBoxLayout()

self.flags\_display\_labels['Z'] = QLabel("Z: F")

self.flags\_display\_labels['N'] = QLabel("N: F")

self.flags\_display\_labels['O'] = QLabel("O: F")

flag\_layout.addWidget(self.flags\_display\_labels['Z'])

flag\_layout.addWidget(self.flags\_display\_labels['N'])

flag\_layout.addWidget(self.flags\_display\_labels['O'])

reg\_flag\_layout.addLayout(flag\_layout)

def update\_gui\_cpu\_status(self):

"""Cập nhật các giá trị của RAM, Registers, và Flags lên GUI."""

# 1. Update RAM Table

for i in range(16):

self.ram\_table.setItem(i, 1, QTableWidgetItem(self.cpu.ram[i]))

# 2. Update Register Labels

self.register\_display\_labels['A'].setText(f"Reg A: {self.cpu.registers['A']} ({self.cpu.registers['A']:08b})")

self.register\_display\_labels['B'].setText(f"Reg B: {self.cpu.registers['B']} ({self.cpu.registers['B']:08b})")

self.register\_display\_labels['IAR'].setText(f"IAR: {self.cpu.registers['IAR']} ({self.cpu.registers['IAR']:04b})")

self.register\_display\_labels['IR\_Opcode'].setText(f"IR (Opcode): {self.cpu.registers['IR\_Opcode']}")

self.register\_display\_labels['IR\_AddrData'].setText(f"IR (Addr/Data): {self.cpu.registers['IR\_AddrData']}")

# 3. Update Flag Labels

self.flags\_display\_labels['Z'].setText(f"Z: {'T' if self.cpu.flags['Z'] else 'F'}")

self.flags\_display\_labels['N'].setText(f"N: {'T' if self.cpu.flags['N'] else 'F'}")

self.flags\_display\_labels['O'].setText(f"O: {'T' if self.cpu.flags['O'] else 'F'}")

def load\_instruction\_gui(self):

"""Hàm xử lý khi nhấn nút 'Load Instruction to RAM'."""

try:

address = int(self.instr\_address\_input.text())

selected\_opcode = self.opcode\_combo.currentData() # Lấy giá trị data (opcode bits)

operand\_text = self.operand\_input.text()

if not (0 <= address <= 15):

QMessageBox.warning(self, "Lỗi đầu vào", "Địa chỉ RAM phải từ 0 đến 15.")

return

if not (len(operand\_text) == 4 and all(c in '01' for c in operand\_text)):

QMessageBox.warning(self, "Lỗi đầu vào", "Operand phải là chuỗi 4 bit nhị phân (ví dụ: 0101).")

return

full\_instruction = selected\_opcode + operand\_text

self.cpu.load\_instruction\_to\_ram(address, full\_instruction)

self.update\_gui\_cpu\_status() # Cập nhật hiển thị RAM sau khi nạp

QMessageBox.information(self, "Thành công", f"Đã nạp lệnh {full\_instruction} vào RAM[{address}]")

except ValueError as e:

QMessageBox.critical(self, "Lỗi đầu vào", f"Dữ liệu không hợp lệ: {e}")

except Exception as e:

QMessageBox.critical(self, "Lỗi", f"Có lỗi xảy ra: {e}")

def step\_cpu(self):

"""Thực hiện một chu kỳ CPU (Fetch-Decode-Execute)."""

if self.cpu.is\_halted:

self.run\_timer.stop()

QMessageBox.information(self, "CPU Halted", "CPU đã dừng thực thi.")

return

if self.cpu.step\_cycle():

# Bước thực hiện thành công, cập nhật GUI

self.update\_gui\_cpu\_status()

# Ở đây bạn sẽ thêm logic cho các đốm sáng và hoạt ảnh

# self.animate\_signal\_path(...)

else:

# CPU báo hiệu đã dừng (ví dụ do HALT hoặc lỗi)

self.run\_timer.stop()

QMessageBox.information(self, "CPU Halted", "CPU đã dừng thực thi.")

def run\_cpu(self):

"""Chạy CPU tự động từng bước."""

if self.cpu.is\_halted:

QMessageBox.information(self, "CPU Halted", "CPU đã dừng. Vui lòng Reset trước khi chạy.")

return

self.run\_timer.start() # Bắt đầu timer để gọi step\_cpu liên tục

self.run\_button.setEnabled(False) # Vô hiệu hóa nút Run khi đang chạy

self.step\_button.setEnabled(False) # Vô hiệu hóa nút Step khi đang chạy

def reset\_cpu(self):

"""Đặt lại trạng thái CPU và GUI."""

self.run\_timer.stop()

self.cpu.reset()

self.update\_gui\_cpu\_status()

self.run\_button.setEnabled(True)

self.step\_button.setEnabled(True)

QMessageBox.information(self, "Reset CPU", "CPU đã được đặt lại trạng thái ban đầu.")

if \_\_name\_\_ == "\_\_main\_\_":

app = QApplication(sys.argv)

window = CPUVisualizerApp()

window.show()

sys.exit(app.exec())

### Giải thích các cập nhật quan trọng:

1. **Cấu trúc GUI (main.py)**:
   * Sử dụng QHBoxLayout làm layout chính để chia cửa sổ thành hai panel: left\_panel (cho controls, RAM, Register) và right\_panel (cho sơ đồ CPU).
   * ram\_table, control\_buttons\_layout, instr\_loader\_layout, register\_and\_flag\_display được thêm vào left\_panel.
2. **Khởi tạo CPU**:
   * self.cpu = CPU() được gọi trong \_\_init\_\_ của CPUVisualizerApp, tạo ra một đối tượng CPU duy nhất để quản lý trạng thái.
3. **Kết nối các nút điều khiển**:
   * self.run\_button.clicked.connect(self.run\_cpu): Khi nhấn nút "Run", phương thức run\_cpu sẽ được gọi.
   * self.step\_button.clicked.connect(self.step\_cpu): Khi nhấn nút "Step", phương thức step\_cpu sẽ được gọi.
   * self.reset\_button.clicked.connect(self.reset\_cpu): Khi nhấn nút "Reset", phương thức reset\_cpu sẽ được gọi.
4. **Hàm update\_gui\_cpu\_status()**:
   * Đây là hàm quan trọng để đồng bộ hóa trạng thái bên trong self.cpu với các hiển thị trên GUI (bảng RAM, giá trị thanh ghi, cờ).
   * Hàm này được gọi sau mỗi thao tác thay đổi trạng thái CPU (như nạp lệnh, chạy step\_cycle, reset).
5. **Hàm load\_instruction\_gui()**:
   * Xử lý việc lấy dữ liệu từ QLineEdit (địa chỉ, operand) và QComboBox (opcode).
   * Sử dụng self.cpu.load\_instruction\_to\_ram() để nạp lệnh vào mô hình CPU.
   * Cập nhật GUI ngay sau đó.
   * Thêm xử lý lỗi cơ bản cho đầu vào người dùng.
6. **Hàm step\_cpu()**:
   * Hàm này gọi self.cpu.step\_cycle(), là hàm thực hiện một chu kỳ Fetch-Decode-Execute.
   * Sau khi CPU thực hiện một bước, nó gọi self.update\_gui\_cpu\_status() để cập nhật hiển thị.
   * Kiểm tra self.cpu.is\_halted để dừng CPU và thông báo khi chương trình kết thúc.
7. **Hàm run\_cpu() và QTimer**:
   * run\_cpu chỉ đơn giản là khởi động self.run\_timer.
   * self.run\_timer được cấu hình để phát tín hiệu timeout mỗi 500ms (0.5 giây).
   * Tín hiệu timeout này được kết nối với self.step\_cpu(), nghĩa là step\_cpu() sẽ tự động được gọi mỗi 0.5 giây khi chế độ "Run" hoạt động, mô phỏng CPU chạy liên tục.
8. **Hàm reset\_cpu()**:
   * Dừng run\_timer (nếu đang chạy).
   * Gọi self.cpu.reset() để đặt lại trạng thái CPU.
   * Cập nhật GUI để phản ánh trạng thái reset.
   * Kích hoạt lại các nút "Run" và "Step".

### Các bước tiếp theo:

1. **Chạy thử**: Chạy lại chương trình python src/main.py. Bạn sẽ thấy giao diện với bảng RAM, các nút, và hiển thị thanh ghi/cờ. Thử nhập một vài lệnh vào RAM và nhấn "Step" hoặc "Run" để xem các giá trị thay đổi.
2. **Trực quan hóa Đốm sáng và Tín hiệu**: Đây là phần thách thức nhất và sẽ yêu cầu sử dụng QGraphicsItem trên QGraphicsScene để vẽ các hình động.
3. **Cập nhật trạng thái trên sơ đồ**: Làm cho các giá trị trên sơ đồ (thanh ghi, IR, flags) thay đổi trực tiếp trên ảnh nền.
4. **Trực Quan Hóa Luồng Xử Lý CPU**

Đây là phần thú vị nhất của dự án, nơi chúng ta sẽ biến mô phỏng logic thành hình ảnh động trực quan. Phần này sẽ phức tạp hơn một chút vì liên quan đến đồ họa và hoạt ảnh trong PyQt6.

### Giai đoạn 4: Tích hợp GUI & Trực quan hóa (Phần 1: Hiển thị trạng thái tĩnh trên sơ đồ)

Để trực quan hóa các "đốm sáng" và cập nhật trạng thái, chúng ta cần:

1. Xác định các vùng trên sơ đồ ảnh của bạn tương ứng với các thành phần CPU (Registers, RAM, IR, IAR, ALU, CU).
2. Tạo các đối tượng đồ họa (QGraphicsTextItem, QGraphicsRectItem) trên QGraphicsScene để hiển thị giá trị và vùng hoạt động.
3. Cập nhật giá trị của các đối tượng này khi trạng thái CPU thay đổi.

#### 1. Xác định Tọa độ trên Sơ đồ (Quan trọng!)

Đây là bước bạn phải tự làm dựa trên hình ảnh cpu\_diagram.jpg của bạn. Bạn cần mở ảnh bằng một công cụ chỉnh sửa ảnh (như Paint, GIMP, Photoshop) hoặc thậm chí một trình duyệt ảnh có khả năng hiển thị tọa độ pixel.

* **Xác định các vùng**: Vẽ một hình chữ nhật bao quanh từng thành phần (Register A, Register B, IR, IAR, một ô RAM, ALU, Control Unit, v.v.).
* **Ghi lại tọa độ**: Ghi lại tọa độ (x, y) của góc trên bên trái và (width, height) của hình chữ nhật đó.
  + Ví dụ:
    - Reg A: (x=100, y=150, width=50, height=20)
    - IAR: (x=50, y=50, width=40, height=20)
    - RAM Cell (Addr 0): (x=700, y=100, width=80, height=20)
* **Xác định các đường dây**: Đối với các đường tín hiệu, bạn cần xác định một chuỗi các điểm (x, y) tạo thành đường dây đó.

Bạn sẽ lưu trữ các tọa độ này trong main.py hoặc một file cấu hình riêng (nhưng để đơn giản, ban đầu chúng ta sẽ đặt trực tiếp vào main.py).

#### 2. Cập nhật file src/main.py

Chúng ta sẽ thêm các QGraphicsTextItem và QGraphicsRectItem vào QGraphicsScene để hiển thị và cập nhật trạng thái.

**File: src/main.py (Cập nhật - chỉ các phần liên quan đến GUI đồ họa)**

Python

import sys

from PyQt6.QtWidgets import (

QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,

QLabel, QPushButton, QTableWidget, QTableWidgetItem, QGraphicsView,

QGraphicsScene, QComboBox, QLineEdit, QFormLayout, QMessageBox, QGraphicsRectItem

)

from PyQt6.QtGui import QPixmap, QColor, QPen, QFont, QBrush

from PyQt6.QtCore import Qt, QTimer, QPointF, QPropertyAnimation, QRectF

from .cpu\_core import CPU

# Định nghĩa TỌA ĐỘ cho các thành phần trên sơ đồ (BẠN CẦN THAY ĐỔI THEO HÌNH CỦA BẠN!)

# Các tọa độ này là giả định, bạn phải thay đổi chúng dựa trên file ảnh của bạn.

# Nên có khoảng cách đủ lớn giữa các thành phần để đặt giá trị

DIAGRAM\_COORDS = {

'IAR\_TEXT\_POS': QPointF(50, 60), # Vị trí chữ số của IAR (Program Counter)

'IR\_OPCODE\_TEXT\_POS': QPointF(200, 60), # Vị trí chữ số Opcode trong IR

'IR\_ADDRDATA\_TEXT\_POS': QPointF(270, 60), # Vị trí chữ số Addr/Data trong IR

'REG\_A\_TEXT\_POS': QPointF(100, 200), # Vị trí chữ số Register A

'REG\_B\_TEXT\_POS': QPointF(100, 250), # Vị trí chữ số Register B

'ALU\_OUT\_TEXT\_POS': QPointF(250, 300), # Vị trí kết quả đầu ra của ALU (ví dụ tạm)

'FLAG\_Z\_POS': QPointF(400, 50), # Vị trí cờ Z

'FLAG\_N\_POS': QPointF(430, 50), # Vị trí cờ N

'FLAG\_O\_POS': QPointF(460, 50), # Vị trí cờ O

# Vị trí các ô RAM (ví dụ cho 3 ô đầu tiên, bạn cần mở rộng cho 16 ô)

# Đây là nơi giá trị RAM sẽ hiển thị trực tiếp trên sơ đồ

'RAM\_CELL\_POSITIONS': {

0: QPointF(700, 100),

1: QPointF(700, 120),

2: QPointF(700, 140),

# ... thêm các địa chỉ khác

},

# Vùng cho các hình chữ nhật bao quanh để highlight

'IAR\_RECT': QRectF(40, 50, 60, 30), # (x, y, width, height)

'IR\_RECT': QRectF(190, 50, 120, 30),

'REG\_A\_RECT': QRectF(90, 190, 70, 30),

'REG\_B\_RECT': QRectF(90, 240, 70, 30),

'ALU\_RECT': QRectF(180, 280, 150, 80), # Ví dụ vùng ALU

'CONTROL\_UNIT\_RECT': QRectF(350, 100, 100, 100), # Ví dụ vùng Control Unit

# Đường dây tín hiệu (sẽ dùng cho animation sau)

# Ví dụ: đường từ RAM đến IR

'PATH\_RAM\_TO\_IR': [QPointF(650, 120), QPointF(320, 70)],

# ... và nhiều đường khác

}

class CPUVisualizerApp(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("CPU Execution Visualizer")

self.setGeometry(100, 100, 1400, 800) # Tăng kích thước cửa sổ

self.cpu = CPU()

self.central\_widget = QWidget()

self.setCentralWidget(self.central\_widget)

self.main\_layout = QHBoxLayout(self.central\_widget)

# Dictionary để lưu trữ các QGraphicsTextItem hiển thị giá trị trên sơ đồ

self.diagram\_text\_items = {}

# Dictionary để lưu trữ các QGraphicsRectItem để highlight vùng

self.diagram\_rect\_highlights = {}

self.init\_ui()

self.update\_gui\_cpu\_status() # Cập nhật trạng thái GUI lần đầu

def init\_ui(self):

# ... (Phần setup left\_panel và right\_panel giống như trước) ...

self.left\_panel = QVBoxLayout()

self.main\_layout.addLayout(self.left\_panel, 1) # Chiếm 1 phần (ví dụ 1/3)

self.right\_panel = QVBoxLayout()

self.main\_layout.addLayout(self.right\_panel, 2) # Chiếm 2 phần (ví dụ 2/3)

# --- Right Panel: CPU Diagram ---

self.scene = QGraphicsScene()

self.view = QGraphicsView(self.scene)

self.right\_panel.addWidget(self.view)

try:

self.cpu\_diagram\_pixmap = QPixmap("assets/image\_212fe2.jpg")

if self.cpu\_diagram\_pixmap.isNull():

QMessageBox.critical(self, "Lỗi tải ảnh", "Không thể tải ảnh sơ đồ CPU. Kiểm tra đường dẫn 'assets/image\_212fe2.jpg'.")

# Fallback to a gray rectangle if image fails to load

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray"))

else:

self.scene.addPixmap(self.cpu\_diagram\_pixmap)

self.scene.setSceneRect(self.cpu\_diagram\_pixmap.rect())

self.view.fitInView(self.scene.sceneRect(), Qt.AspectRatioMode.KeepAspectRatio)

except Exception as e:

QMessageBox.critical(self, "Lỗi tải ảnh", f"Xảy ra lỗi khi tải ảnh: {e}")

self.scene.addRect(0, 0, 800, 400, QPen(QColor("red")), QColor("lightgray")) # Fallback

# --- Thêm các thành phần hiển thị trên SƠ ĐỒ CPU ---

self.setup\_diagram\_elements()

# --- Left Panel: Controls and Data --- (Giữ nguyên như trước)

# 1. Control Buttons (Run, Step, Reset)

self.control\_buttons\_layout = QHBoxLayout()

self.run\_button = QPushButton("Run")

self.step\_button = QPushButton("Step")

self.reset\_button = QPushButton("Reset")

self.control\_buttons\_layout.addWidget(self.run\_button)

self.control\_buttons\_layout.addWidget(self.step\_button)

self.control\_buttons\_layout.addWidget(self.reset\_button)

self.left\_panel.addLayout(self.control\_buttons\_layout)

self.run\_button.clicked.connect(self.run\_cpu)

self.step\_button.clicked.connect(self.step\_cpu)

self.reset\_button.clicked.connect(self.reset\_cpu)

# 2. RAM Table

self.ram\_group\_box = QWidget()

self.ram\_group\_box\_layout = QVBoxLayout(self.ram\_group\_box)

self.left\_panel.addWidget(self.ram\_group\_box)

self.ram\_table = QTableWidget(16, 2)

self.ram\_table.setHorizontalHeaderLabels(["Address", "Data (8-bit)"])

for i in range(16):

self.ram\_table.setItem(i, 0, QTableWidgetItem(str(i)))

self.ram\_table.setItem(i, 1, QTableWidgetItem("00000000"))

self.ram\_table.setColumnWidth(0, 50)

self.ram\_table.setColumnWidth(1, 90)

self.ram\_table.verticalHeader().setVisible(False)

self.ram\_group\_box\_layout.addWidget(QLabel("RAM (Instruction/Data)"))

self.ram\_group\_box\_layout.addWidget(self.ram\_table)

# 3. Instruction Loader

self.instr\_loader\_layout = QFormLayout()

self.left\_panel.addLayout(self.instr\_loader\_layout)

self.instr\_address\_input = QLineEdit("0")

self.instr\_address\_input.setPlaceholderText("0-15")

self.instr\_address\_input.setFixedWidth(50)

self.opcode\_combo = QComboBox()

for opcode\_bits, info in self.cpu.opcode\_map.items():

self.opcode\_combo.addItem(f"{info['name']} ({opcode\_bits})", opcode\_bits)

self.operand\_input = QLineEdit("0000")

self.operand\_input.setPlaceholderText("4-bit binary (e.g., 0101)")

self.operand\_input.setFixedWidth(80)

self.load\_instr\_button = QPushButton("Load Instruction to RAM")

self.load\_instr\_button.clicked.connect(self.load\_instruction\_gui)

self.instr\_loader\_layout.addRow(QLabel("Địa chỉ RAM:"), self.instr\_address\_input)

self.instr\_loader\_layout.addRow(QLabel("Opcode:"), self.opcode\_combo)

self.instr\_loader\_layout.addRow(QLabel("Operand/Addr:"), self.operand\_input)

self.instr\_loader\_layout.addRow(self.load\_instr\_button)

# 4. Register & Flag Display (trên bảng điều khiển bên trái)

self.register\_display\_labels = {}

self.flags\_display\_labels = {}

self.setup\_register\_and\_flag\_display()

# Timer cho chế độ Run

self.run\_timer = QTimer(self)

self.run\_timer.setInterval(500)

self.run\_timer.timeout.connect(self.step\_cpu)

def setup\_diagram\_elements(self):

"""Thiết lập các QGraphicsItem để hiển thị giá trị và highlight trên sơ đồ CPU."""

font = QFont("Arial", 10, QFont.Weight.Bold)

# IAR (Program Counter)

self.diagram\_text\_items['IAR'] = self.scene.addText("IAR: 0", font)

self.diagram\_text\_items['IAR'].setPos(DIAGRAM\_COORDS['IAR\_TEXT\_POS'])

self.diagram\_rect\_highlights['IAR'] = self.scene.addRect(DIAGRAM\_COORDS['IAR\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

# IR (Instruction Register) - Opcode and Addr/Data parts

self.diagram\_text\_items['IR\_Opcode'] = self.scene.addText("Op:0000", font)

self.diagram\_text\_items['IR\_Opcode'].setPos(DIAGRAM\_COORDS['IR\_OPCODE\_TEXT\_POS'])

self.diagram\_text\_items['IR\_AddrData'] = self.scene.addText("Ad:0000", font)

self.diagram\_text\_items['IR\_AddrData'].setPos(DIAGRAM\_COORDS['IR\_ADDRDATA\_TEXT\_POS'])

self.diagram\_rect\_highlights['IR'] = self.scene.addRect(DIAGRAM\_COORDS['IR\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

# Registers A & B

self.diagram\_text\_items['A'] = self.scene.addText("A: 0 (00)", font)

self.diagram\_text\_items['A'].setPos(DIAGRAM\_COORDS['REG\_A\_TEXT\_POS'])

self.diagram\_rect\_highlights['A'] = self.scene.addRect(DIAGRAM\_COORDS['REG\_A\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

self.diagram\_text\_items['B'] = self.scene.addText("B: 0 (00)", font)

self.diagram\_text\_items['B'].setPos(DIAGRAM\_COORDS['REG\_B\_TEXT\_POS'])

self.diagram\_rect\_highlights['B'] = self.scene.addRect(DIAGRAM\_COORDS['REG\_B\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

# Flags (Z, N, O)

self.diagram\_text\_items['Z\_flag'] = self.scene.addText("Z: F", font)

self.diagram\_text\_items['Z\_flag'].setPos(DIAGRAM\_COORDS['FLAG\_Z\_POS'])

self.diagram\_text\_items['N\_flag'] = self.scene.addText("N: F", font)

self.diagram\_text\_items['N\_flag'].setPos(DIAGRAM\_COORDS['FLAG\_N\_POS'])

self.diagram\_text\_items['O\_flag'] = self.scene.addText("O: F", font)

self.diagram\_text\_items['O\_flag'].setPos(DIAGRAM\_COORDS['FLAG\_O\_POS'])

# RAM Cells on Diagram

self.diagram\_ram\_cells = {} # Lưu trữ QGraphicsTextItem cho mỗi ô RAM trên sơ đồ

for i in range(16):

if i in DIAGRAM\_COORDS['RAM\_CELL\_POSITIONS']:

text\_item = self.scene.addText(f"RAM[{i}]: 00000000", QFont("Arial", 8))

text\_item.setPos(DIAGRAM\_COORDS['RAM\_CELL\_POSITIONS'][i])

self.diagram\_ram\_cells[i] = text\_item

# Highlight for ALU (will be activated during ADD/SUB)

self.diagram\_rect\_highlights['ALU'] = self.scene.addRect(DIAGRAM\_COORDS['ALU\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

# Highlight for Control Unit (will be activated during Decode/Execute phases)

self.diagram\_rect\_highlights['CONTROL\_UNIT'] = self.scene.addRect(DIAGRAM\_COORDS['CONTROL\_UNIT\_RECT'], QPen(Qt.GlobalColor.transparent), QBrush(Qt.GlobalColor.transparent))

def update\_gui\_cpu\_status(self):

"""Cập nhật các giá trị của RAM, Registers, và Flags lên GUI (cả bảng và sơ đồ)."""

# 1. Update RAM Table (bên trái)

for i in range(16):

self.ram\_table.setItem(i, 1, QTableWidgetItem(self.cpu.ram[i]))

# Cập nhật RAM cells trên sơ đồ

if i in self.diagram\_ram\_cells:

self.diagram\_ram\_cells[i].setPlainText(f"RAM[{i}]: {self.cpu.ram[i]}")

# 2. Update Register Labels (bên trái)

self.register\_display\_labels['A'].setText(f"Reg A: {self.cpu.registers['A']} ({self.cpu.registers['A']:08b})")

self.register\_display\_labels['B'].setText(f"Reg B: {self.cpu.registers['B']} ({self.cpu.registers['B']:08b})")

self.register\_display\_labels['IAR'].setText(f"IAR: {self.cpu.registers['IAR']} ({self.cpu.registers['IAR']:04b})")

self.register\_display\_labels['IR\_Opcode'].setText(f"IR (Opcode): {self.cpu.registers['IR\_Opcode']}")

self.register\_display\_labels['IR\_AddrData'].setText(f"IR (Addr/Data): {self.cpu.registers['IR\_AddrData']}")

# 3. Update Register/IR/IAR Labels ON DIAGRAM

self.diagram\_text\_items['IAR'].setPlainText(f"IAR: {self.cpu.registers['IAR']:04b}") # IAR display as 4-bit binary

self.diagram\_text\_items['IR\_Opcode'].setPlainText(f"Op:{self.cpu.registers['IR\_Opcode']}")

self.diagram\_text\_items['IR\_AddrData'].setPlainText(f"Ad:{self.cpu.registers['IR\_AddrData']}")

self.diagram\_text\_items['A'].setPlainText(f"A: {self.cpu.registers['A']:08b}") # Reg A display as 8-bit binary

self.diagram\_text\_items['B'].setPlainText(f"B: {self.cpu.registers['B']:08b}") # Reg B display as 8-bit binary

# 4. Update Flag Labels (bên trái và trên sơ đồ)

z\_flag\_status = 'T' if self.cpu.flags['Z'] else 'F'

n\_flag\_status = 'T' if self.cpu.flags['N'] else 'F'

o\_flag\_status = 'T' if self.cpu.flags['O'] else 'F'

self.flags\_display\_labels['Z'].setText(f"Z: {z\_flag\_status}")

self.flags\_display\_labels['N'].setText(f"N: {n\_flag\_status}")

self.flags\_display\_labels['O'].setText(f"O: {o\_flag\_status}")

self.diagram\_text\_items['Z\_flag'].setPlainText(f"Z: {z\_flag\_status}")

self.diagram\_text\_items['N\_flag'].setPlainText(f"N: {n\_flag\_status}")

self.diagram\_text\_items['O\_flag'].setPlainText(f"O: {o\_flag\_status}")

# Reset tất cả các highlight

for rect\_item in self.diagram\_rect\_highlights.values():

rect\_item.setPen(QPen(Qt.GlobalColor.transparent))

rect\_item.setBrush(QBrush(Qt.GlobalColor.transparent))

# ... (Các hàm load\_instruction\_gui, step\_cpu, run\_cpu, reset\_cpu giữ nguyên,

# nhưng step\_cpu và reset\_cpu sẽ cần gọi thêm hàm reset\_highlights) ...

def reset\_cpu(self):

"""Đặt lại trạng thái CPU và GUI."""

self.run\_timer.stop()

self.cpu.reset()

self.update\_gui\_cpu\_status()

self.run\_button.setEnabled(True)

self.step\_button.setEnabled(True)

QMessageBox.information(self, "Reset CPU", "CPU đã được đặt lại trạng thái ban đầu.")

# Thêm việc reset highlight khi reset CPU

for rect\_item in self.diagram\_rect\_highlights.values():

rect\_item.setPen(QPen(Qt.GlobalColor.transparent))

rect\_item.setBrush(QBrush(Qt.GlobalColor.transparent))

def highlight\_component(self, component\_key, color=Qt.GlobalColor.yellow):

"""Hàm dùng để highlight một thành phần trên sơ đồ."""

if component\_key in self.diagram\_rect\_highlights:

self.diagram\_rect\_highlights[component\_key].setPen(QPen(color, 2))

self.diagram\_rect\_highlights[component\_key].setBrush(QBrush(QColor(color.red(), color.green(), color.blue(), 50))) # Màu nền trong suốt

# Tự động tắt highlight sau một khoảng thời gian ngắn

QTimer.singleShot(200, lambda: self.reset\_highlight(component\_key))

def reset\_highlight(self, component\_key):

"""Đặt lại highlight của một thành phần."""

if component\_key in self.diagram\_rect\_highlights:

self.diagram\_rect\_highlights[component\_key].setPen(QPen(Qt.GlobalColor.transparent))

self.diagram\_rect\_highlights[component\_key].setBrush(QBrush(Qt.GlobalColor.transparent))

### Giải thích các cập nhật mới:

1. **DIAGRAM\_COORDS**:
   * Đây là một từ điển mà bạn **phải tự điền tọa độ chính xác** dựa trên hình ảnh sơ đồ CPU của bạn. Tôi đã đưa ra các giá trị giả định.
   * QPointF dùng cho vị trí của chữ (góc trên bên trái).
   * QRectF dùng cho vùng hình chữ nhật bao quanh một thành phần, sẽ được sử dụng để highlight.
2. **self.diagram\_text\_items và self.diagram\_rect\_highlights**:
   * Hai từ điển này được tạo trong \_\_init\_\_ để lưu trữ các đối tượng QGraphicsTextItem (hiển thị văn bản) và QGraphicsRectItem (dùng để highlight) mà chúng ta thêm vào QGraphicsScene.
3. **setup\_diagram\_elements()**:
   * Hàm mới này được gọi trong init\_ui để tạo và thêm tất cả các QGraphicsTextItem và QGraphicsRectItem ban đầu vào self.scene.
   * **QGraphicsTextItem**: Dùng để hiển thị các giá trị (IAR, IR, Reg A/B, Flags, RAM cell). Bạn đặt vị trí của chúng bằng setPos(DIAGRAM\_COORDS['KEY\_POS']).
   * **QGraphicsRectItem**: Được tạo cho mỗi thành phần mà bạn muốn highlight (IAR, IR, Reg A/B, ALU, Control Unit). Ban đầu chúng được đặt trong suốt (transparent).
4. **Cập nhật update\_gui\_cpu\_status()**:
   * Ngoài việc cập nhật bảng RAM và các label bên trái, hàm này giờ đây còn cập nhật setPlainText() của các QGraphicsTextItem tương ứng trên sơ đồ.
   * **Reset Highlights**: Một vòng lặp nhỏ được thêm vào đầu hàm để reset tất cả các highlight về trạng thái trong suốt trước khi một bước mới bắt đầu, đảm bảo chỉ có thành phần đang hoạt động được highlight.
5. **Hàm highlight\_component() và reset\_highlight()**:
   * highlight\_component(component\_key, color): Khi bạn gọi hàm này với một component\_key (ví dụ: 'A', 'IR', 'ALU'), nó sẽ tìm QGraphicsRectItem tương ứng và đặt màu viền (pen) và màu nền (brush) để làm nổi bật.
   * QTimer.singleShot(200, ...): Được sử dụng để tự động tắt highlight sau 200ms, tạo hiệu ứng nhấp nháy.
   * reset\_highlight(component\_key): Đặt lại màu viền và màu nền của highlight về trong suốt.

### Các bước tiếp theo:

* **Chạy thử**: Chạy chương trình. Bây giờ, khi bạn nhấn "Step" hoặc "Run", ngoài các giá trị bên trái thay đổi, các giá trị trên sơ đồ cũng sẽ cập nhật. Các highlight sẽ nhấp nháy nhưng chưa có logic để kích hoạt đúng lúc.
* **Trực quan hóa Đốm sáng/Tín hiệu (Phần 2: Hoạt ảnh động)**: Đây là phần phức tạp nhất, nơi chúng ta sẽ tạo các đối tượng hình tròn nhỏ di chuyển theo các đường dây và kích hoạt chúng tại các pha Fetch, Decode, Execute.
* **Kích hoạt Highlight đúng lúc**: Cần bổ sung các lệnh gọi self.highlight\_component() vào step\_cpu hoặc trong các hàm fetch\_instruction, decode\_instruction, execute\_instruction của cpu\_core.py để làm nổi bật thành phần đang hoạt động.

Đây là phần cốt lõi để làm cho chương trình của bạn trở nên sống động. Chúng ta sẽ tập trung vào việc tạo các "đốm sáng" di chuyển trên sơ đồ và kích hoạt việc làm nổi bật các thành phần CPU tương ứng với từng pha trong chu trình lệnh.

### Giai đoạn 4: Tích hợp GUI & Trực quan hóa (Phần 2: Hoạt ảnh động và Highlight)

Để thực hiện phần này, chúng ta cần:

1. **Định nghĩa các đường dây tín hiệu**: Đây là một bước thủ công **rất quan trọng** mà bạn phải làm dựa trên sơ đồ CPU của mình.
2. **Tạo một hàm/lớp hỗ trợ cho hoạt ảnh đốm sáng**: Để dễ dàng tạo và điều khiển các đốm sáng.
3. **Điều chỉnh logic chu kỳ CPU**: Để kích hoạt các hoạt ảnh và highlight tại đúng thời điểm.

#### 1. Định nghĩa Đường dây Tín hiệu (DIAGRAM\_COORDS trong main.py)

Bạn cần mở file ảnh cpu\_diagram.jpg trong một trình chỉnh sửa ảnh (ví dụ: Paint trên Windows) và ghi lại các tọa độ pixel cho các đường dây.

* Mỗi đường dây sẽ là một danh sách các QPointF. Hoạt ảnh đốm sáng sẽ di chuyển tuần tự qua các điểm này.
* Bạn cần xác định các đường dây quan trọng tương ứng với các pha:
  + **Fetch**: PC -> Bus Địa chỉ -> RAM -> Bus Dữ liệu -> IR
  + **Decode**: IR -> Control Unit
  + **Execute**: (Tùy lệnh)
    - LOAD\_A/B: RAM -> Bus Dữ liệu -> Reg A/B
    - STORE\_A/B: Reg A/B -> Bus Dữ liệu -> RAM
    - ADD/SUB: Reg A, Reg B -> ALU -> Reg A
    - JUMP: Địa chỉ đích -> PC

**Cập nhật file: src/main.py** (Thêm vào trong từ điển DIAGRAM\_COORDS)

Python

# src/main.py (Phần DIAGRAM\_COORDS)

# ... (các định nghĩa tọa độ đã có) ...

# Định nghĩa CÁC ĐƯỜNG DÂY TÍN HIỆU (Các điểm QPointF tạo thành đường)

# CÁC TỌA ĐỘ NÀY LÀ VÍ DỤ! BẠN PHẢI THAY THẾ BẰNG TỌA ĐỘ TỪ SƠ ĐỒ CỦA BẠN.

SIGNAL\_PATHS = {

# Fetch Phase

'IAR\_TO\_RAM\_ADDR\_BUS': [QPointF(60, 70), QPointF(500, 70), QPointF(500, 100), QPointF(650, 100)], # IAR -> Address Bus -> RAM

'RAM\_DATA\_TO\_IR': [QPointF(750, 150), QPointF(550, 150), QPointF(550, 70), QPointF(320, 70)], # RAM Data Out -> Data Bus -> IR

# Decode Phase

'IR\_TO\_CONTROL\_UNIT': [QPointF(250, 80), QPointF(250, 150), QPointF(370, 150)], # IR -> Control Unit

# Execute Phase - LOAD (A/B)

'RAM\_DATA\_TO\_REG\_A': [QPointF(750, 150), QPointF(550, 150), QPointF(550, 200), QPointF(180, 200)], # RAM Data Out -> Data Bus -> Reg A

'RAM\_DATA\_TO\_REG\_B': [QPointF(750, 150), QPointF(550, 150), QPointF(550, 250), QPointF(180, 250)], # RAM Data Out -> Data Bus -> Reg B

# Execute Phase - STORE (A/B)

'REG\_A\_TO\_RAM\_DATA\_BUS': [QPointF(180, 200), QPointF(550, 200), QPointF(550, 150), QPointF(750, 150)], # Reg A -> Data Bus -> RAM

'REG\_B\_TO\_RAM\_DATA\_BUS': [QPointF(180, 250), QPointF(550, 250), QPointF(550, 150), QPointF(750, 150)], # Reg B -> Data Bus -> RAM

# Execute Phase - ALU Operations (ADD/SUB)

'REG\_A\_TO\_ALU': [QPointF(170, 200), QPointF(200, 200), QPointF(200, 280)], # Reg A -> ALU Input

'REG\_B\_TO\_ALU': [QPointF(170, 250), QPointF(200, 250), QPointF(200, 280)], # Reg B -> ALU Input

'ALU\_TO\_REG\_A': [QPointF(250, 350), QPointF(250, 200), QPointF(170, 200)], # ALU Output -> Reg A

# Execute Phase - JUMP (update IAR)

'ADDR\_TO\_IAR\_JUMP': [QPointF(290, 70), QPointF(290, 40), QPointF(70, 40), QPointF(70, 60)] # IR Addr -> IAR

}

#### 2. Tạo lớp SignalAnimator (trong file src/gui\_elements.py mới)

Chúng ta sẽ tạo một file mới để quản lý các đối tượng đồ họa động, giữ cho main.py gọn gàng hơn.

**File: src/gui\_elements.py**

Python

# src/gui\_elements.py

from PyQt6.QtWidgets import QGraphicsEllipseItem

from PyQt6.QtGui import QColor, QBrush, QPainterPath, QPen

from PyQt6.QtCore import QPropertyAnimation, QPointF, QEasingCurve, Qt, QObject, pyqtSignal

class SignalAnimator(QObject):

# Định nghĩa một signal để thông báo khi animation kết thúc

animation\_finished = pyqtSignal()

def \_\_init\_\_(self, scene):

super().\_\_init\_\_()

self.scene = scene

self.active\_animations = [] # Để theo dõi các animation đang chạy

def animate\_path(self, path\_points, color=Qt.GlobalColor.blue, dot\_size=8, duration=500):

"""

Tạo và chạy hoạt ảnh một đốm sáng di chuyển dọc theo một đường dẫn.

path\_points: list của QPointF

color: Màu của đốm sáng

dot\_size: Kích thước của đốm sáng (đường kính)

duration: Thời gian hoạt ảnh (ms)

"""

if not path\_points or len(path\_points) < 2:

print("Lỗi: Đường dẫn hoạt ảnh không hợp lệ (cần ít nhất 2 điểm).")

return

dot = QGraphicsEllipseItem(0, 0, dot\_size, dot\_size)

dot.setBrush(QBrush(color))

dot.setPen(QPen(Qt.GlobalColor.black, 0.5)) # Viền mỏng

self.scene.addItem(dot)

dot.setZValue(100) # Đảm bảo đốm sáng nằm trên các thành phần khác

# Tạo QPainterPath từ list các QPointF

path = QPainterPath()

path.moveTo(path\_points[0])

for p in path\_points[1:]:

path.lineTo(p)

animation = QPropertyAnimation(dot, b"pos", self)

animation.setDuration(duration)

animation.setStartValue(path\_points[0]) # Start value

# PyQt6 QPropertyAnimation for QGraphicsItem.pos doesn't directly take QPainterPath.

# We need to set key values along the path.

# This is a simplification; for complex paths, you might need to

# manually calculate intermediate points or use a custom property.

# Simple path animation for now (only two points, or just start/end)

# For actual path following, more advanced techniques are needed or rely on 'pos' property

# and manually update it based on path interpolation, or just use keyframes.

# A workaround for animating along QPainterPath with QPropertyAnimation (more complex, not native)

# For this guide, let's simplify to direct point-to-point animation

# or just set start/end values for a "line" animation.

# Let's create key values for each segment in the path

# This is a common way to simulate path animation

segment\_duration = duration / (len(path\_points) - 1) if len(path\_points) > 1 else duration

for i in range(len(path\_points)):

animation.setKeyValueAt(i / (len(path\_points) - 1.0), path\_points[i])

animation.setEasingCurve(QEasingCurve.Type.Linear)

# Kết nối sự kiện finished để xóa đốm sáng và giải phóng tài nguyên

animation.finished.connect(lambda: self.\_cleanup\_animation(dot, animation))

self.active\_animations.append(animation) # Thêm vào danh sách theo dõi

animation.start()

# Nếu chỉ có một đốm sáng duy nhất và cần thông báo khi nó hoàn thành:

animation.finished.connect(self.animation\_finished)

return animation

def \_cleanup\_animation(self, dot\_item, animation\_object):

"""Dọn dẹp sau khi hoạt ảnh kết thúc."""

self.scene.removeItem(dot\_item)

if animation\_object in self.active\_animations:

self.active\_animations.remove(animation\_object)

animation\_object.deleteLater() # Quan trọng để giải phóng bộ nhớ

dot\_item.deleteLater()

**Lưu ý về SignalAnimator:**

* QPropertyAnimation có thể không trực tiếp lấy QPainterPath cho thuộc tính pos của QGraphicsItem. Cách tôi triển khai ở trên (setKeyValueAt) là một cách để mô phỏng việc di chuyển qua nhiều điểm. Đối với các đường phức tạp, bạn có thể cần một phương pháp nội suy phức tạp hơn hoặc tạo một thuộc tính tùy chỉnh.
* animation.finished.connect(lambda: self.\_cleanup\_animation(dot, animation)) đảm bảo rằng đốm sáng và hoạt ảnh sẽ tự động được xóa khỏi scene và bộ nhớ sau khi hoàn thành.
* animation\_finished = pyqtSignal(): Đây là một tín hiệu tùy chỉnh sẽ phát ra khi MỘT animation hoàn thành. Bạn có thể sử dụng nó để điều khiển luồng thực thi trong step\_cpu.

#### 3. Cập nhật file src/main.py (Kích hoạt hoạt ảnh và highlight)

Bây giờ chúng ta sẽ sửa đổi step\_cpu để thực hiện từng pha (Fetch, Decode, Execute) một cách tuần tự, với các hoạt ảnh và highlight tương ứng. Điều này đòi hỏi step\_cpu phải giữ trạng thái (đang ở pha nào) và sử dụng QTimer để chuyển từ pha này sang pha khác.

**File: src/main.py (Cập nhật)**

Python

# src/main.py

# ... (các import đã có) ...

from .cpu\_core import CPU

from .gui\_elements import SignalAnimator # Import lớp SignalAnimator

# ... (DIAGRAM\_COORDS và SIGNAL\_PATHS đã định nghĩa) ...

class CPUVisualizerApp(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# ... (khởi tạo window, cpu) ...

self.cpu = CPU()

self.signal\_animator = SignalAnimator(self.scene) # Khởi tạo animator

# ... (khởi tạo GUI) ...

self.cpu\_current\_phase = 'IDLE' # Trạng thái hiện tại của CPU cycle (IDLE, FETCH, DECODE, EXECUTE)

self.cpu\_phase\_timer = QTimer(self) # Timer để điều khiển các pha

self.cpu\_phase\_timer.setInterval(200) # Thời gian giữa các bước/pha (có thể điều chỉnh)

self.cpu\_phase\_timer.timeout.connect(self.\_advance\_cpu\_phase) # Kết nối tới hàm xử lý pha

self.animation\_step\_duration = 500 # Thời gian cho mỗi hoạt ảnh tín hiệu

# ... (các hàm init\_ui, setup\_diagram\_elements, setup\_register\_and\_flag\_display, update\_gui\_cpu\_status, load\_instruction\_gui) ...

def \_clear\_all\_highlights\_and\_animations(self):

"""Xóa tất cả highlights và dừng/dọn dẹp animations."""

for rect\_item in self.diagram\_rect\_highlights.values():

rect\_item.setPen(QPen(Qt.GlobalColor.transparent))

rect\_item.setBrush(QBrush(Qt.GlobalColor.transparent))

# Đảm bảo tất cả animations cũ được dừng và dọn dẹp

for anim in self.signal\_animator.active\_animations[:]: # Lặp trên bản sao để tránh lỗi khi sửa đổi list

anim.stop()

# \_cleanup\_animation sẽ được gọi qua signal, nhưng đảm bảo mọi thứ sạch sẽ

if anim.targetObject(): # Kiểm tra targetObject có tồn tại không

self.scene.removeItem(anim.targetObject())

anim.deleteLater()

self.signal\_animator.active\_animations.clear()

def highlight\_component(self, component\_key, color=Qt.GlobalColor.yellow, duration=200):

"""Hàm dùng để highlight một thành phần trên sơ đồ và tự động tắt."""

if component\_key in self.diagram\_rect\_highlights:

rect\_item = self.diagram\_rect\_highlights[component\_key]

original\_pen = rect\_item.pen()

original\_brush = rect\_item.brush()

rect\_item.setPen(QPen(color, 2))

rect\_item.setBrush(QBrush(QColor(color.red(), color.green(), color.blue(), 50)))

# Dùng QTimer để tắt highlight sau một khoảng thời gian

QTimer.singleShot(duration, lambda: self.\_reset\_single\_highlight(component\_key, original\_pen, original\_brush))

def \_reset\_single\_highlight(self, component\_key, original\_pen, original\_brush):

if component\_key in self.diagram\_rect\_highlights:

self.diagram\_rect\_highlights[component\_key].setPen(original\_pen)

self.diagram\_rect\_highlights[component\_key].setBrush(original\_brush)

def \_animate\_signal(self, path\_name, color=Qt.GlobalColor.red):

"""Gọi SignalAnimator để chạy hoạt ảnh trên đường dây."""

if path\_name in SIGNAL\_PATHS:

self.signal\_animator.animate\_path(SIGNAL\_PATHS[path\_name], color, duration=self.animation\_step\_duration)

else:

print(f"Lỗi: Đường dẫn tín hiệu '{path\_name}' không được định nghĩa.")

def step\_cpu(self):

"""Kích hoạt chu kỳ CPU từng bước (kiểm soát pha)."""

if self.cpu.is\_halted:

self.run\_timer.stop()

QMessageBox.information(self, "CPU Halted", "CPU đã dừng thực thi.")

self.cpu\_current\_phase = 'IDLE' # Đặt lại pha

return

# Dọn dẹp highlight và animation cũ trước khi bắt đầu pha mới

self.\_clear\_all\_highlights\_and\_animations()

# Bắt đầu chu kỳ pha, hoặc tiếp tục nếu đang ở giữa

if self.cpu\_current\_phase == 'IDLE':

self.cpu\_current\_phase = 'FETCH'

# Dừng run\_timer và bắt đầu cpu\_phase\_timer để điều khiển từng pha

self.run\_timer.stop()

self.cpu\_phase\_timer.start() # Bắt đầu timer cho các pha

self.run\_button.setEnabled(False) # Tạm thời vô hiệu hóa các nút điều khiển chính

self.step\_button.setEnabled(False)

def \_advance\_cpu\_phase(self):

"""Tiến lên một pha trong chu kỳ CPU."""

if self.cpu.is\_halted:

self.cpu\_phase\_timer.stop()

self.cpu\_current\_phase = 'IDLE'

self.run\_button.setEnabled(True)

self.step\_button.setEnabled(True)

return

if self.cpu\_current\_phase == 'FETCH':

print("--- PHASE: FETCH ---")

self.highlight\_component('IAR', Qt.GlobalColor.red)

self.\_animate\_signal('IAR\_TO\_RAM\_ADDR\_BUS', Qt.GlobalColor.red)

# Sau khi animation IAR\_TO\_RAM\_ADDR\_BUS kết thúc, tiếp tục pha

# Để thực sự đợi animation, chúng ta sẽ kết nối signal animation\_finished

# Tuy nhiên, để đơn giản hóa cho hướng dẫn, chúng ta sẽ dùng timer

# và giả định animation đủ nhanh để chuyển pha.

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50) # Đợi animation xong

self.cpu.fetch\_instruction() # Thực hiện logic fetch

self.update\_gui\_cpu\_status() # Cập nhật GUI ngay lập tức sau logic

# Animation phản hồi từ RAM đến IR (sau khi fetch)

self.highlight\_component('IR', Qt.GlobalColor.red)

self.\_animate\_signal('RAM\_DATA\_TO\_IR', Qt.GlobalColor.green)

self.cpu\_current\_phase = 'DECODE'

elif self.cpu\_current\_phase == 'DECODE':

print("--- PHASE: DECODE ---")

self.cpu\_phase\_timer.setInterval(200) # Đặt lại thời gian cho pha decode

self.highlight\_component('IR', Qt.GlobalColor.darkYellow)

self.highlight\_component('CONTROL\_UNIT', Qt.GlobalColor.cyan)

self.\_animate\_signal('IR\_TO\_CONTROL\_UNIT', Qt.GlobalColor.darkMagenta) # Tín hiệu giải mã

decoded\_instruction = self.cpu.decode\_instruction() # Thực hiện logic decode

# self.update\_gui\_cpu\_status() # Cập nhật GUI (thường không có thay đổi lớn ở pha này)

self.cpu\_current\_phase = 'EXECUTE'

self.cpu.last\_decoded\_instruction = decoded\_instruction # Lưu lại để execute

elif self.cpu\_current\_phase == 'EXECUTE':

print("--- PHASE: EXECUTE ---")

self.cpu\_phase\_timer.stop() # Dừng timer pha, chuyển giao quyền điều khiển lại cho step/run

decoded\_instruction = self.cpu.last\_decoded\_instruction # Lấy lệnh đã giải mã

# Kích hoạt highlight và animation theo từng loại lệnh

op\_name = decoded\_instruction['name']

if op\_name == 'ADD' or op\_name == 'SUB':

self.highlight\_component('ALU', Qt.GlobalColor.magenta)

self.highlight\_component('A', Qt.GlobalColor.green)

self.highlight\_component('B', Qt.GlobalColor.green)

self.\_animate\_signal('REG\_A\_TO\_ALU', Qt.GlobalColor.darkCyan)

self.\_animate\_signal('REG\_B\_TO\_ALU', Qt.GlobalColor.darkCyan)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration \* 2) # Đợi 2 animations

self.cpu\_phase\_timer.start() # Chạy lại timer để đợi animation xong

self.cpu\_current\_phase = 'EXECUTE\_ALU\_RESULT' # Chờ kết quả ALU

return # Thoát để chờ timer

elif op\_name == 'LOAD\_A':

self.highlight\_component('A', Qt.GlobalColor.green)

self.\_animate\_signal('RAM\_DATA\_TO\_REG\_A', Qt.GlobalColor.darkCyan)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM'

return

# ... Thêm các điều kiện cho các lệnh khác (LOAD\_B, STORE\_A/B, JUMP, etc.)

elif op\_name == 'LOAD\_B':

self.highlight\_component('B', Qt.GlobalColor.green)

self.\_animate\_signal('RAM\_DATA\_TO\_REG\_B', Qt.GlobalColor.darkCyan)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM'

return

elif op\_name == 'STORE\_A':

self.highlight\_component('A', Qt.GlobalColor.darkGreen)

self.\_animate\_signal('REG\_A\_TO\_RAM\_DATA\_BUS', Qt.GlobalColor.darkRed)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM'

return

elif op\_name == 'STORE\_B':

self.highlight\_component('B', Qt.GlobalColor.darkGreen)

self.\_animate\_signal('REG\_B\_TO\_RAM\_DATA\_BUS', Qt.GlobalColor.darkRed)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM'

return

elif op\_name == 'JUMP' or op\_name == 'JUMP\_NEG' or op\_name == 'JUMP\_ZERO':

self.highlight\_component('IAR', Qt.GlobalColor.blue)

self.\_animate\_signal('ADDR\_TO\_IAR\_JUMP', Qt.GlobalColor.blue)

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM'

return

# Đối với NOP, HALT và các lệnh không có animation đặc biệt

self.\_finish\_execute\_phase()

elif self.cpu\_current\_phase == 'EXECUTE\_ALU\_RESULT': # Pha phụ cho kết quả ALU

print("--- PHASE: EXECUTE (ALU Result) ---")

self.\_animate\_signal('ALU\_TO\_REG\_A', Qt.GlobalColor.red)

self.highlight\_component('A', Qt.GlobalColor.red) # Reg A nhận kết quả

self.cpu\_phase\_timer.setInterval(self.animation\_step\_duration + 50)

self.cpu\_phase\_timer.start()

self.cpu\_current\_phase = 'EXECUTE\_WAIT\_ANIM' # Chờ animation kết thúc

return # Đảm bảo không execute() ngay lập tức

elif self.cpu\_current\_phase == 'EXECUTE\_WAIT\_ANIM': # Pha chờ animation kết thúc

print("--- PHASE: EXECUTE (Waiting for Animation) ---")

# Sau khi chờ animation, thực hiện lệnh và kết thúc chu kỳ

self.cpu.execute\_instruction(self.cpu.last\_decoded\_instruction) # Thực hiện logic execute

self.update\_gui\_cpu\_status() # Cập nhật GUI sau khi execute

self.\_finish\_execute\_phase() # Chuyển sang pha kết thúc chu kỳ

self.update\_gui\_cpu\_status() # Cập nhật lần cuối sau pha

def \_finish\_execute\_phase(self):

"""Kết thúc pha Execute và chuẩn bị cho chu kỳ tiếp theo."""

self.cpu\_phase\_timer.stop() # Dừng timer pha

self.cpu\_current\_phase = 'IDLE' # Đặt lại về IDLE

self.run\_button.setEnabled(True) # Kích hoạt lại các nút

self.step\_button.setEnabled(True)

# Nếu đang ở chế độ "Run", run\_timer sẽ tự động gọi step\_cpu lại

if self.run\_button.isEnabled() and not self.cpu.is\_halted: # Kiểm tra để tránh lỗi khi run\_button bị disable

if not self.run\_timer.isActive(): # Nếu timer chưa chạy lại (sau step hoặc sau lần đầu run)

self.run\_timer.start() # Tự động gọi step\_cpu sau interval

def run\_cpu(self):

"""Chạy CPU tự động từng bước."""

if self.cpu.is\_halted:

QMessageBox.information(self, "CPU Halted", "CPU đã dừng. Vui lòng Reset trước khi chạy.")

return

self.run\_timer.start() # Bắt đầu timer để gọi step\_cpu liên tục

self.run\_button.setEnabled(False) # Vô hiệu hóa nút Run khi đang chạy

self.step\_button.setEnabled(False) # Vô hiệu hóa nút Step khi đang chạy

self.step\_cpu() # Bắt đầu chu kỳ đầu tiên ngay lập tức

def reset\_cpu(self):

"""Đặt lại trạng thái CPU và GUI."""

self.run\_timer.stop()

self.cpu\_phase\_timer.stop()

self.cpu\_current\_phase = 'IDLE' # Reset pha

self.\_clear\_all\_highlights\_and\_animations() # Xóa tất cả animation và highlight

self.cpu.reset()

self.update\_gui\_cpu\_status()

self.run\_button.setEnabled(True)

self.step\_button.setEnabled(True)

QMessageBox.information(self, "Reset CPU", "CPU đã được đặt lại trạng thái ban đầu.")

### Giải thích các cập nhật quan trọng:

1. **SIGNAL\_PATHS**:
   * Bạn cần **cẩn thận điền tọa độ chính xác** cho các đường dây tín hiệu trong từ điển này. Mỗi giá trị là một list các QPointF mà đốm sáng sẽ di chuyển qua.
   * Các path\_name (ví dụ: 'IAR\_TO\_RAM\_ADDR\_BUS') sẽ được sử dụng để gọi hoạt ảnh tương ứng.
2. **SignalAnimator (trong gui\_elements.py)**:
   * Hàm animate\_path giờ đây tạo một QGraphicsEllipseItem (đốm sáng) và sử dụng QPropertyAnimation để di chuyển nó.
   * Nó sẽ tự động xóa đốm sáng khỏi scene và giải phóng bộ nhớ khi hoạt ảnh kết thúc thông qua animation.finished.connect.
3. **Quản lý Pha CPU (self.cpu\_current\_phase và self.cpu\_phase\_timer)**:
   * self.cpu\_current\_phase: Biến trạng thái để theo dõi CPU đang ở pha nào (IDLE, FETCH, DECODE, EXECUTE, v.v.).
   * self.cpu\_phase\_timer: Thay vì run\_timer tự động gọi step\_cpu, giờ đây step\_cpu chỉ khởi động cpu\_phase\_timer.
   * \_advance\_cpu\_phase(): Hàm này được cpu\_phase\_timer gọi định kỳ. Nó chứa logic cho từng pha:
     + **FETCH**: Kích hoạt highlight IAR và IR, chạy hoạt ảnh từ IAR đến RAM, rồi từ RAM đến IR. Sau đó gọi self.cpu.fetch\_instruction().
     + **DECODE**: Kích hoạt highlight IR và Control Unit, chạy hoạt ảnh từ IR đến Control Unit. Gọi self.cpu.decode\_instruction().
     + **EXECUTE**: Đây là pha phức tạp nhất. Tùy thuộc vào lệnh đã giải mã, nó sẽ:
       - Kích hoạt highlight các thanh ghi/ALU liên quan.
       - Chạy các hoạt ảnh đường tín hiệu phù hợp (ví dụ: Regs đến ALU, RAM đến Reg, v.v.).
       - Sử dụng các pha phụ như EXECUTE\_ALU\_RESULT hoặc EXECUTE\_WAIT\_ANIM và điều chỉnh cpu\_phase\_timer.setInterval để tạo độ trễ, cho phép hoạt ảnh hoàn thành trước khi chuyển sang bước logic tiếp theo.
       - Cuối cùng gọi self.cpu.execute\_instruction() và cập nhật GUI.
4. **\_clear\_all\_highlights\_and\_animations()**: Đảm bảo tất cả các highlight và animation cũ được dọn dẹp trước khi một chu kỳ mới hoặc một pha mới bắt đầu.
5. **Cải thiện highlight\_component**: Giờ đây nó tự động tắt sau một duration ngắn, tạo hiệu ứng nhấp nháy.
6. **\_animate\_signal()**: Hàm tiện ích để gọi SignalAnimator.animate\_path() dễ dàng hơn.

Bây giờ, khi bạn chạy chương trình, nhấn "Step" hoặc "Run", bạn sẽ thấy CPU tiến qua các pha, các thành phần được highlight và các "đốm sáng" di chuyển trên các đường dây tín hiệu.

Đây là phần phức tạp nhất của dự án. Nó đòi hỏi sự kiên nhẫn để điều chỉnh tọa độ và thời gian hoạt ảnh cho khớp với sơ đồ của bạn. Bạn sẽ cần thử nghiệm nhiều để cân chỉnh self.animation\_step\_duration và cpu\_phase\_timer.setInterval cho mượt mà.

1. **Content**
2. **Content**
3. **Content**
4. **Content**
5. **Content**
6. **Content**
7. **Content**
8. **Content**
9. **Content**