Chapter 6 - Digital Circuit Simulator

Design, Features, and Specifications

Digital logic circuits are currently the most common form of electronic circuits used in computers, cell phones, tablets, cars, digital radios, and most other electronic applications. Digital circuits consist of "logic" gates, memory, and clock circuits that when combined perform a high level function such as a microprocessor or digital radio. At first glance creating a digital simulator in Python seems rather trivial since the programming language has conditional statements and boolean variables such as `and`, `or`, `not`, `True`, and `False`. The challenge for this project is to design a real-time clock system that can operate efficiently and achieve perhaps greater than 100 Hz operation on a desktop computer.

Approach:

- Modify the Diagram Editor App to create the Digital Circuit Simulator
- KISS - Keep it simple, silly
- DRY - Don't repeat yourself
- SOC - Separation of concerns
- User-interface
  - TopFrame class
  - File menu frame
  - Settings menu frame
  - LED settings frame
  - Rotation button
  - Help menu frame
  - Left Frame Class with Circuit Component Menu
  - Canvas class
  - Mouse class
- Circuit components
  - AND gate
  - OR gate
  - NAND gate
  - NOR gate
  - NOT gate

- XOR gate
- XNOR gate
- Wire Class
- Clock class
- Grid class
- Switch class
- LED class
  - LED size options: large or small
  - LED color options: red, yellow, blue, green
- Text Class

- Analysis
  - Combinational Logic Simulation
  - Sequential Logic Simulation
  - Circuit traversal algorithm
- Counter Circuit Simulation
  - 74LS161 Synchronous 4-Bit Counter
  - 74LS273 Octal D Flip-Flop with Clear
  - 28C16 16K (2K x 8) Parallel EEPROMs
  - 7-Segment Display

Key Technologies Needed:

- File save & load in json format - multiple lists? - Component list and connection list

Project Setup

Language: Python 3.11
IDE: PyCharm 2023.2.1 (Community Edition)
Project directory: D:/EETools/DigitalSimulator
Graphics library: CustomTkinter (https://customtkinter.tomschimansky.com/)

External libraries:

- pip install customtkinter
- python.exe -m pip install --upgrade pip
- pip install ctkcolorpicker
- pip install tkinter-tooltip

- pip install pyInstaller - Create .exe file
- Add images and icons directories to the project.

Digital Components

Reference:

- "Digital Computer Electronics", A. P. Malvino, Ph. D. and J. A. Brown, Glencoe/McGraw-Hill, 3rd Edition, 1999, Internet Archive PDF

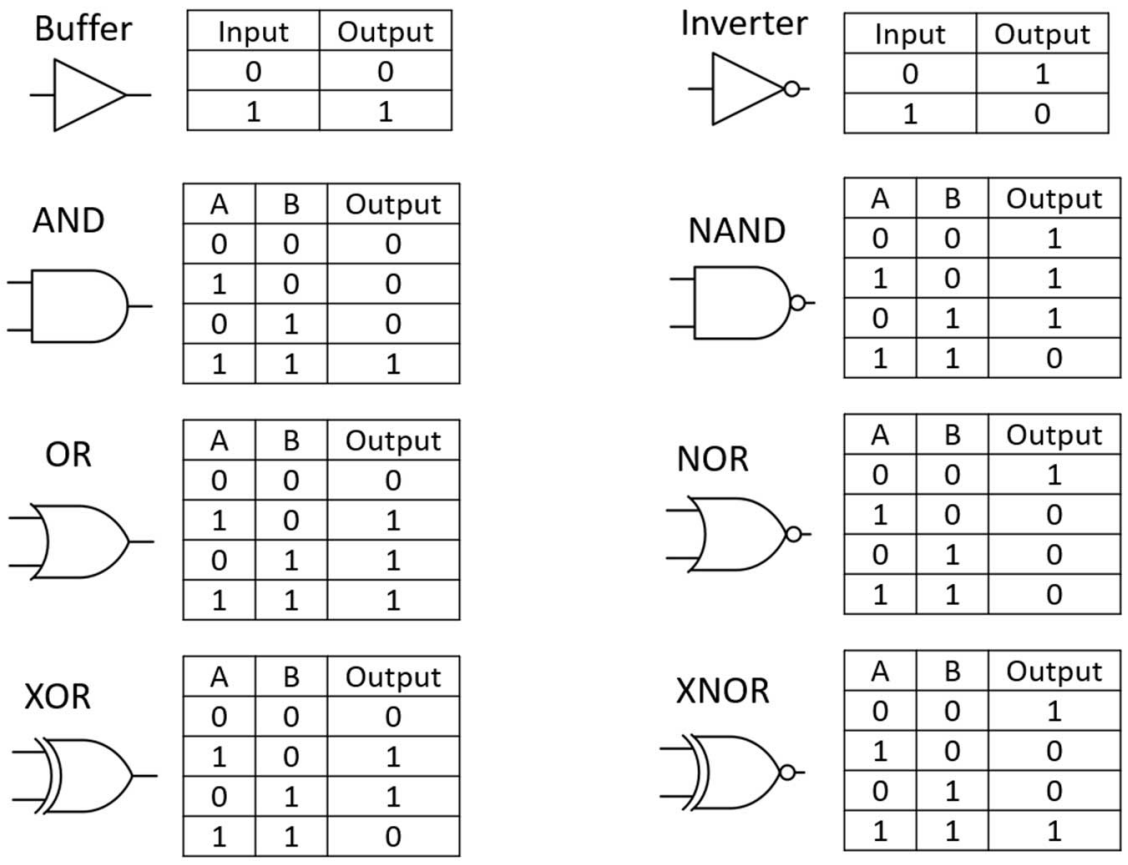


Image courtesy of Global Science Network

Digital logic gates can easily be represented in Python where inputs are True or False. The output is True or False.

```
output = input1 and input2  # Model for AND gate
```

Unfortunately, the canvas shape methods cannot draw a closed curved polygon. We will need to use image files for the graphical representation of the logic gates. We will use a set

of image files courtesy of Logix.

Clock Component

For synchronous circuit simulation, we need a real-time clock simulator that generates a continuous set of pulses that toggle between 1 and 0 or True and False. We will use the threading library to run a continuous clock signal on a separate thread.

```
import threading
import time

state = False


def background_calculation():
    # set the time
    time.sleep(1)

    # Toggle and print state
    global state
    state = not state
    print(state)
    background_calculation()


def main():
    thread = threading.Thread(target=background_calculation)
    thread.start()


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

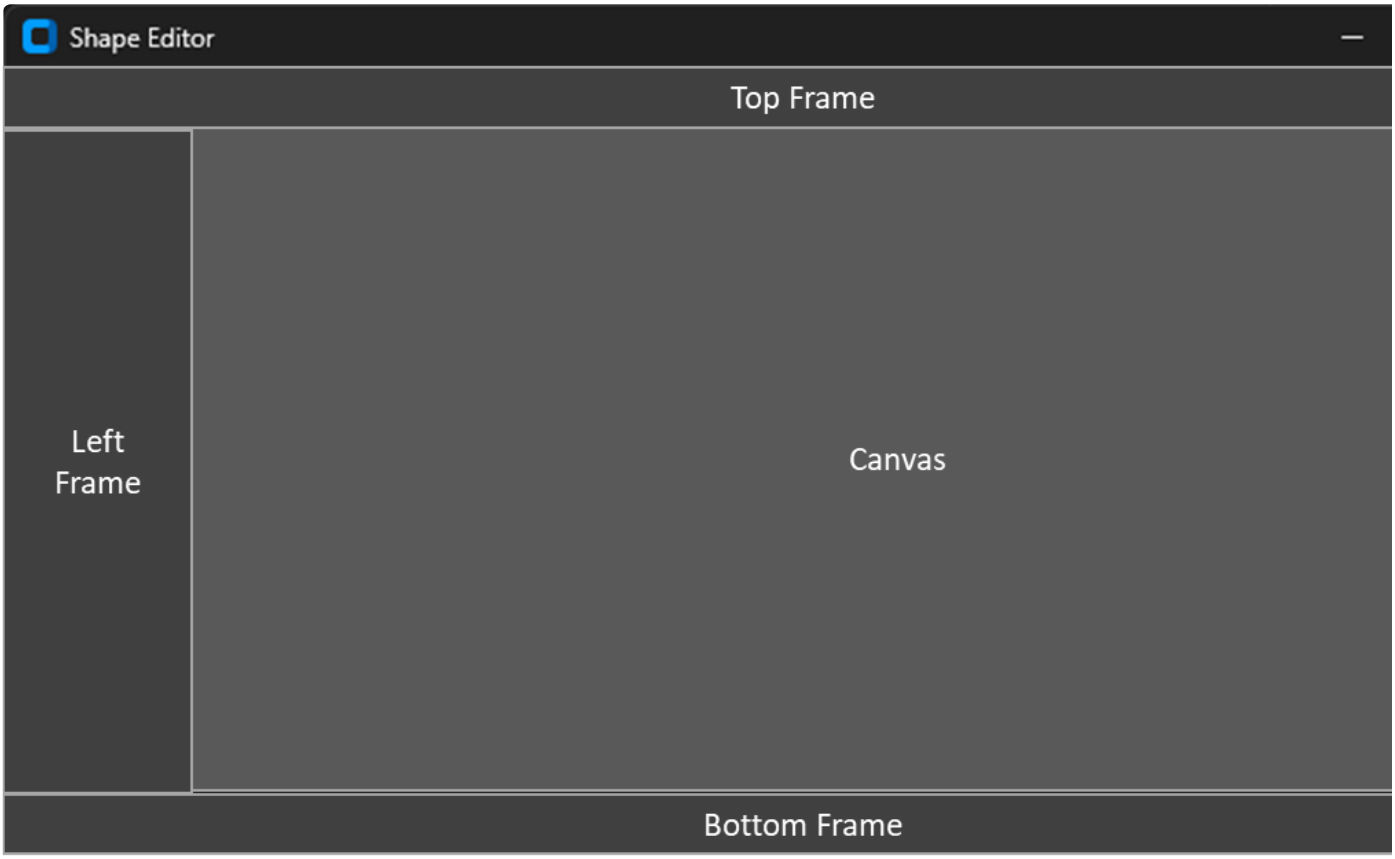
Console Output

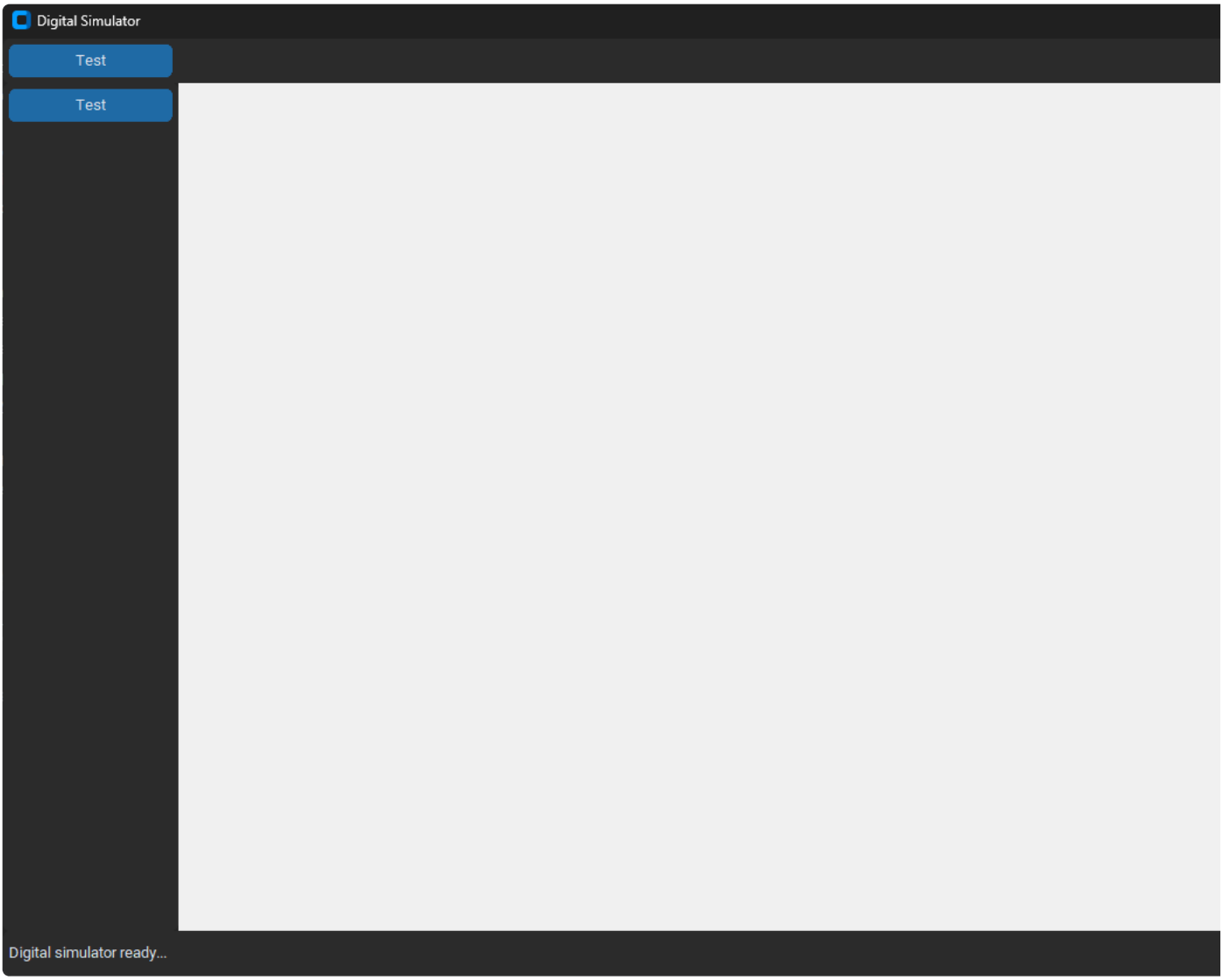
```
True
False
True
False
True
```

```
False
True
False
True
False
True
```

Main User Interface

The initial user interface design is shown in the image below. It consists of a CustomTkinter window with a canvas widget and three custom frame widgets. Custom widgets will allow us to create a modular UI design so that the main file class is small and manageable.





digital_simulator.py

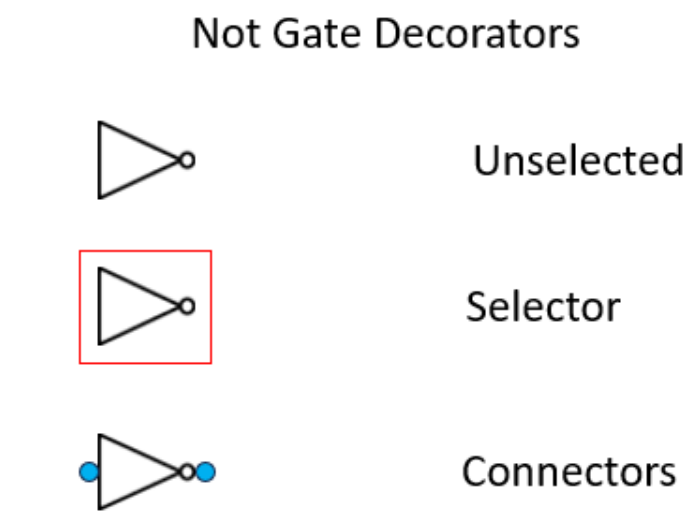
```
import customtkinter as ctk

ctk.set_appearance_mode("System")  # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue")  # Themes: "blue" (standard), "green",
"dark-blue"


class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100")  # w, h, x, y
        self.title("Digital Simulator")
```

```
        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Test")
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

        left_frame_button = ctk.CTkButton(self.left_frame, text="Test")
        left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

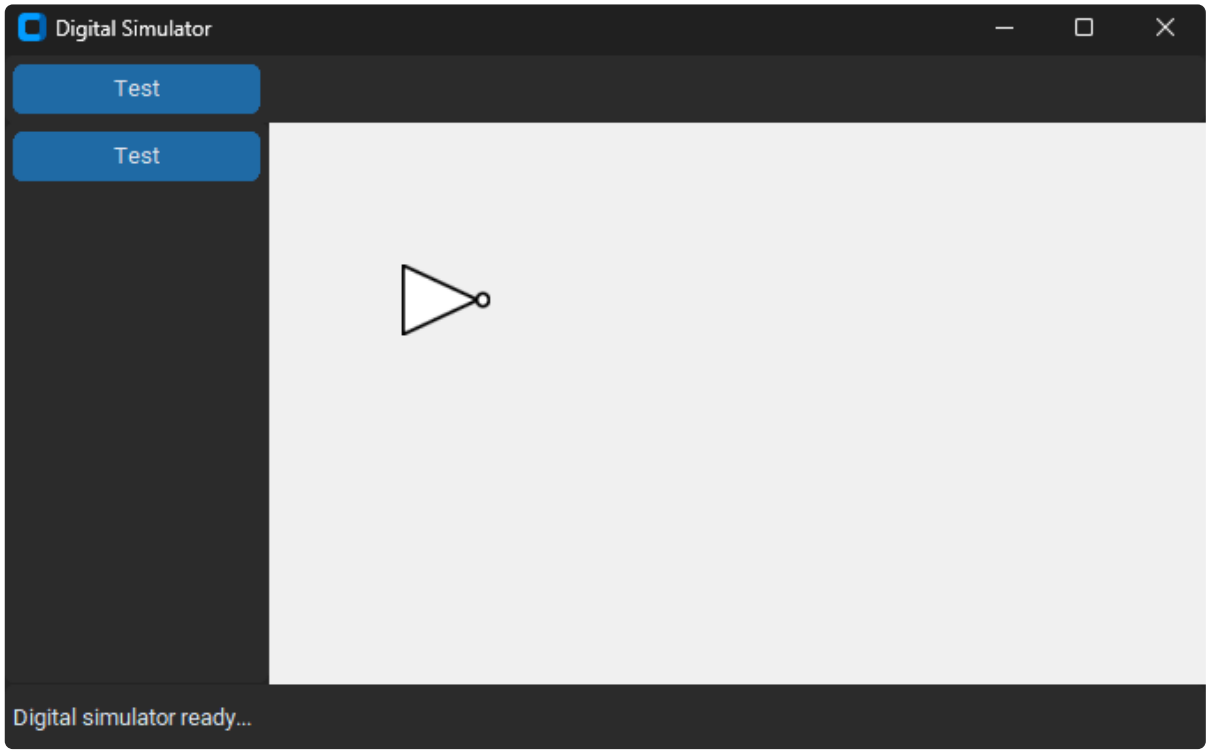
Digital Gate Design

Objectives:

- Create a Not Gate class
- Gate is an image file
- Add selector - shown when the gate is selected
- Add connectors - shown when drawing wires
- Draw gate on canvas from a left frame button



Not Gate Class



Comp_Lib/not_gate.py

```
import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk
```

```
class NotGate:
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')
```

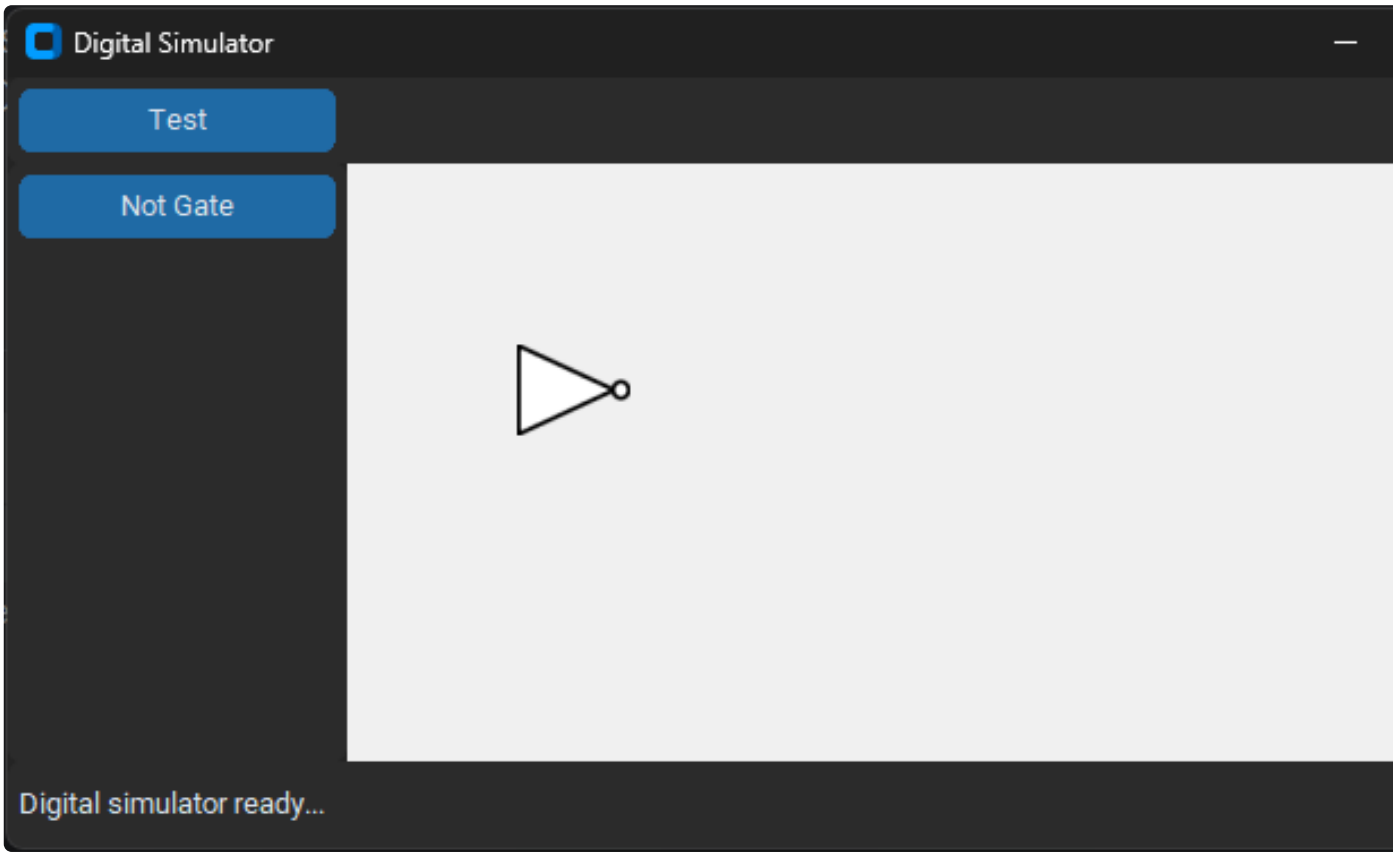
digital_simulator.py

```
import customtkinter as ctk
from Comp_Lib import NotGate  # Added Not gate import here

. . .

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Create test canvas widget here
        self.gate = NotGate(self.canvas, 100, 100)  # Created not gate object
here


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
```

```
    app = DigitalSimulatorApp()
    app.mainloop()
```

Note: The not gate object must be created using `self.gate` not `gate` or it will be deleted by the Python garbage collector and will not be shown on the screen when the program is run.

Create a Not gate from a left frame button

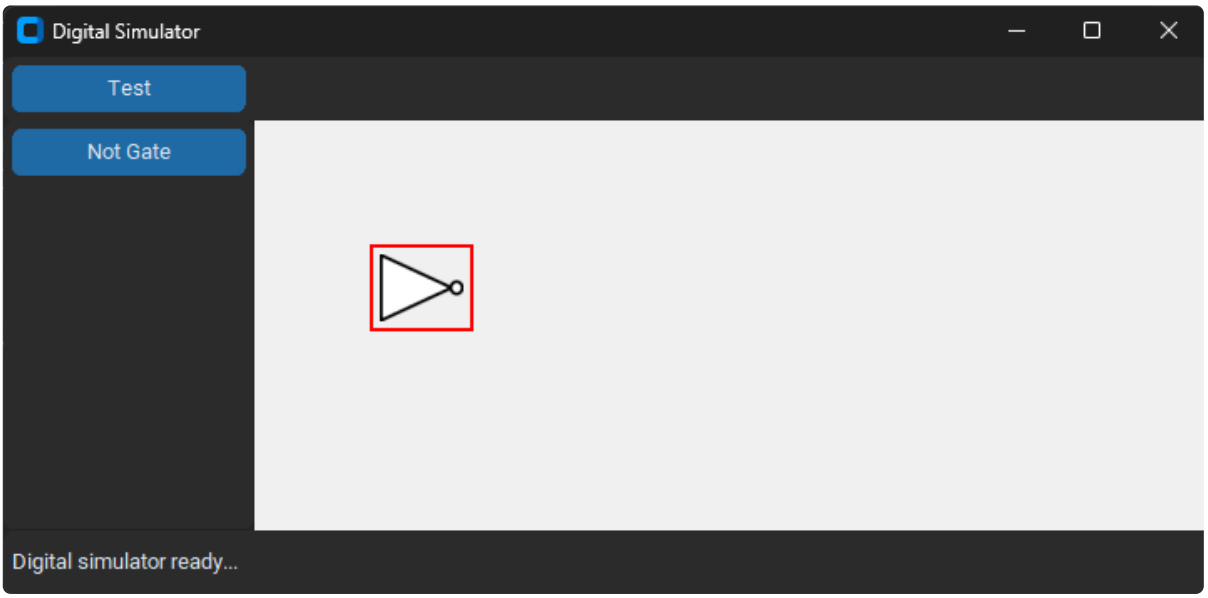


digital_simulator.py

```
. . .

        left_frame_button = ctk.CTkButton(self.left_frame, text="Not Gate",
command=self.create_not_gate)  # Aded call to button handler
        left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)

. . .
```

```
# Create test canvas widget here
        # self.gate = NotGate(self.canvas, 100, 100)
        self.gate = None  # Variable to store the gate object

    def create_not_gate(self):  # Added new method
        self.gate = NotGate(self.canvas, 100, 100)


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

Add selector to Not Gate class

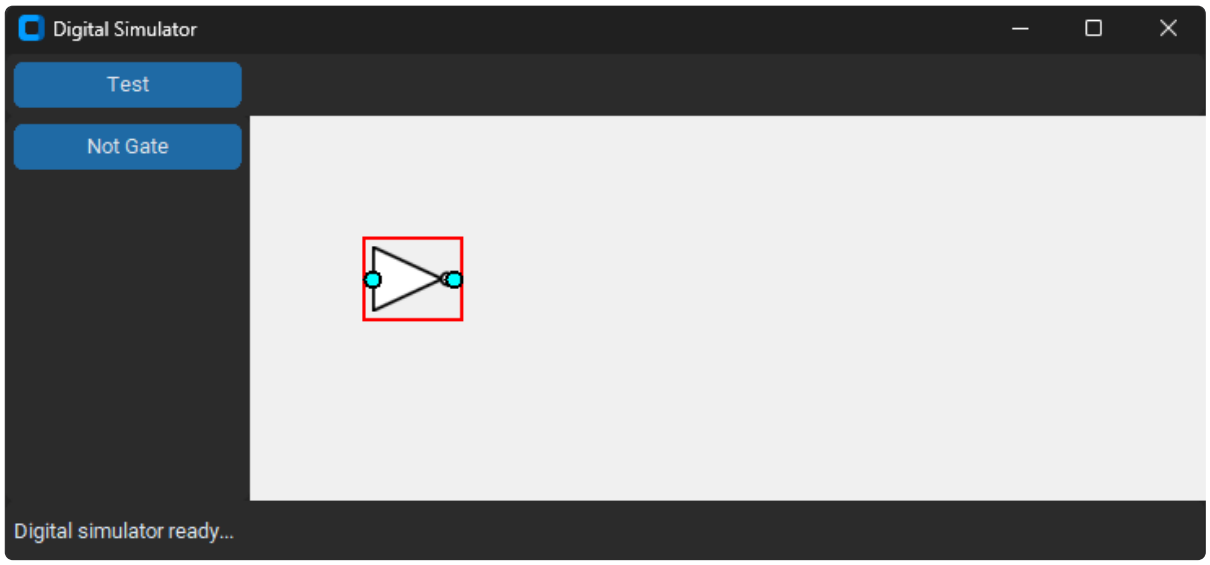


Comp_Lib/not_gate.py

```
import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk


class NotGate:
    def __init__(self, canvas, x1, y1):
```

```
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None  # Added variable for selector id
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0

        self.is_selected = True  # Added boolean variable for gate selection

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()  # Added call to update bbox
        self.create_selector()  # Added call to create the selector

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

    def create_selector(self):  # Added method to create selector
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)

    def update_bbox(self):  # Added method to update the bounding box (bbox) to
the gate's current location
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)
```

Not Gate Connectors



New Connector Class

Connector class features:

- Given the canvas and coordinates the class initializer creates a cyan oval at the x, y coordinates
- update() method updates the connector position using the `canvas.coords()` method
- `set_position()` method that sets the connector coordinates to provided x, y coordinates
- Connector hit test method to determine if the provided x, y coordinates are within the bounds of the connector

Wire_Lib/connector.py

```
class Connector:
    def __init__(self, canvas, name, x, y):
        """Connector class"""
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y

        self.id = None

        self.radius = 5
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
self.radius,
                                              self.x + self.radius, self.y +
```

```
self.radius)

        self.create_connector()

    def create_connector(self):
        # Create the connector here
        points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
        self.id = self.canvas.create_oval(points, fill="cyan", outline="black",
width=2, tags='connector')

    def update(self):
        """Update the connector here"""
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
self.radius,
                                              self.x + self.radius, self.y +
self.radius)
        points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
        self.canvas.coords(self.id, points)

    def set_pos(self, x, y):
        """Set the connector position here"""
        self.x = x
        self.y = y

    def connector_hit_test(self, x, y):
        """Connector hit test"""
        if self.x1 <= x <= self.x2 and self.y1 <= y <= self.y2:
            return True
        else:
            return False

    def __repr__(self):
        return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
                " (" + str(self.x2) + ", " + str(self.y2) + ")")
```

New Point Class

Point class features:

- Create points with .x and .y notation

- Eliminates the need for [0] and [1] on point lists or `_x` or `_y` notation on variable names

Helper_Lib/point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Comp_Lib/not_gate.py

```
import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk

from Wire_Lib import Connector  # Added import for the Connector class
from Helper_Lib import Point # Added import for the new Point Class

. . .

        self.angle = 0

        self.is_selected = True
        self.is_drawing = True  # Added boolean that is set when a wire is
being draw

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.in1_id, self.out1_id = None, None  # Added
        self.conn_list = []  # Add a connector list
        self.create_connectors()  # Added call to create connectors

. . .
```

```
    def create_connectors(self):  # Added new method to create connectors
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
center.y)  # Out1
        self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
        self.conn_list = [self.out1_id, self.in1_id]
```

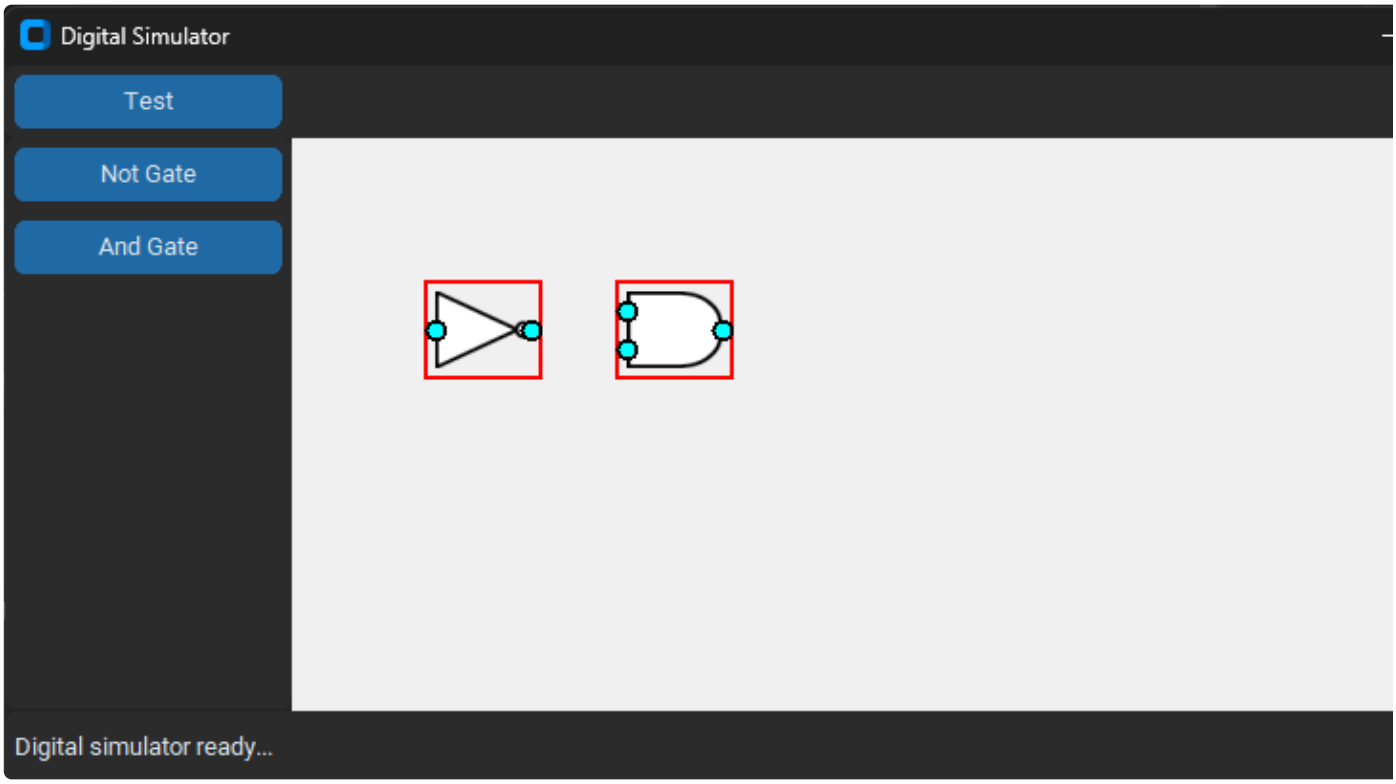
Notes:

- Connectors are positioned at the input and output points on the Not gate symbol
- Positions are calculated relative to the center of the Not gate dimensions
- Connectors have names that will be used later for wire connections

And Gate Class

Objectives:

- Create a new And Gate Class
- Create a new Comp Base Class for And Gate and Not Gate (for DRY)
- Create common selector and connectors for 2-input gates
- Let Not Gate override connectors for 1-input gate
- Draw And Gate from Left Frame Button



New Component Base class called Comp

Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk

from Helper_Lib import Point
from Wire_Lib.connector import Connector


class Comp:
    def __init__(self, canvas, x1, y1):
        """Base class for gate classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
```

```
        self.is_selected = True
        self.is_drawing = True

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]
```

Comp_Lib/not_gate.py

```
from pathlib import Path

from Comp_Lib.component import Comp
from Wire_Lib import Connector
from Helper_Lib import Point


class NotGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.in1_id, self.out1_id = None, None
        self.create_connectors()

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)

    def create_connectors(self):
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
center.y)  # Out1
        self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
        self.conn_list = [self.out1_id, self.in1_id]
```

Comp_Lib/and_gate.py

```
from pathlib import Path

from Comp_Lib.component import Comp


class AndGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/and_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
```

Notes:

- And gate class is very simple (for KISS)
- Base class has the methods used to create the gate
- The class basically sets the gate image file and calls common methods to create the gate

Move Gates with Mouse

Objectives:

- Select a component on the canvas
- Move the component with the left mouse button
- Update the component based on the mouse position

New Canvas Class

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

    def redraw(self):
        for c in self.comp_list:
            c.update()
```

New Mouse Class

UI_Lib/mouse.py

```
from Helper_Lib import Point


class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_mouse_bind_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)

        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                x2, y2 = self.selected_comp.x2, self.selected_comp.y2
                self.offset1.set(x - x1, y - y1)
                self.offset2.set(x - x2, y - y2)
            else:
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                # self.offset1.x, self.offset1.y =
self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
                self.offset1.set(x - x1, y - y1)

    def move_left_drag(self, event):
        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1 = event.x - self.offset1.x
                y1 = event.y - self.offset1.y
                # x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
                x2 = event.x - self.offset2.x
                y2 = event.y - self.offset2.y
```

```
                # x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
                self.selected_comp.x1, self.selected_comp.y1 = x1, y1
                self.selected_comp.x2, self.selected_comp.y2 = x2, y2
                self.canvas.redraw()
            else:
                x = event.x - self.offset1.x
                y = event.y - self.offset1.y
                # x, y = self.canvas.grid.snap_to_grid(x, y)
                self.selected_comp.x1, self.selected_comp.y1 = x, y
                self.canvas.redraw()

    def move_left_up(self, _event):
        if self.selected_comp:
            self.offset1.set(0, 0)
            self.offset2.set(0, 0)

    def select_hit_test(self, x, y):
        for s in self.canvas.comp_list:
            if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
                # print("Shape hit: ", s)
                self.selected_comp = s
                s.is_selected = True
                self.canvas.redraw()
                return

        # No shape hit - unselect all
        self.selected_comp = None
        self.unselect_all()

    def unselect_all(self):
        for s in self.canvas.comp_list:
            s.is_selected = False
        self.canvas.redraw()
```

Add updates to Not Gate and And Gate Classes

Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk


class Comp:
```

```
    def __init__(self, canvas, x1, y1):
        """Base class for gate classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
        self.filename = None

        self.is_selected = False
        self.is_drawing = False

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

    def update_position(self):
        """Update the position when the gate object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1)  # Update position

    def update_image(self, filename):
        """Update the image for gate symbol rotation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)  # Update
image rotation
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.canvas.itemconfig(self.id, image=self.ph_image)  # Update image

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
```

```
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
        self.set_selector_visibility()

    def update_selector(self):
        """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.canvas.coords(self.sel_id, x1, y1, x2, y2)
        self.set_selector_visibility()

    def set_selector_visibility(self):
        """Set the selector visibility state"""
        if self.is_selected:
            self.canvas.itemconfig(self.sel_id, state='normal')
        else:
            self.canvas.itemconfig(self.sel_id, state='hidden')

    def set_connector_visibility(self):
        """Set the connector visibility state"""
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')
```

Comp_Lib/not_gate.py

```
from pathlib import Path

from Comp_Lib.component import Comp
from Wire_Lib import Connector
from Helper_Lib import Point


class NotGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
```

```
        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.in1_id, self.out1_id = None, None
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
center.y)  # Out1
        self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
        self.conn_list = [self.out1_id, self.in1_id]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0 or self.angle == 180:
            sign_x = lambda x: -1 if (self.angle == 180) else 1  # Lambda
function to set sign of x
            self.out1_id.set_pos(center.x + sign_x(self.angle) * w / 2,
center.y)
            self.in1_id.set_pos(center.x - sign_x(self.angle) * w / 2,
center.y)
```

```
        elif self.angle == 90 or self.angle == 270:
            sign_y = lambda y: -1 if (self.angle == 270) else 1  # Lambda
function to set sign of y
            self.out1_id.set_pos(center.x, center.y - sign_y(self.angle) * h /
2)
            self.in1_id.set_pos(center.x, center.y + sign_y(self.angle) * h /
2)

        for c in self.conn_list:
            c.update()

        # self.move_connected_wires()
```

Comp_Lib/and_gate.py

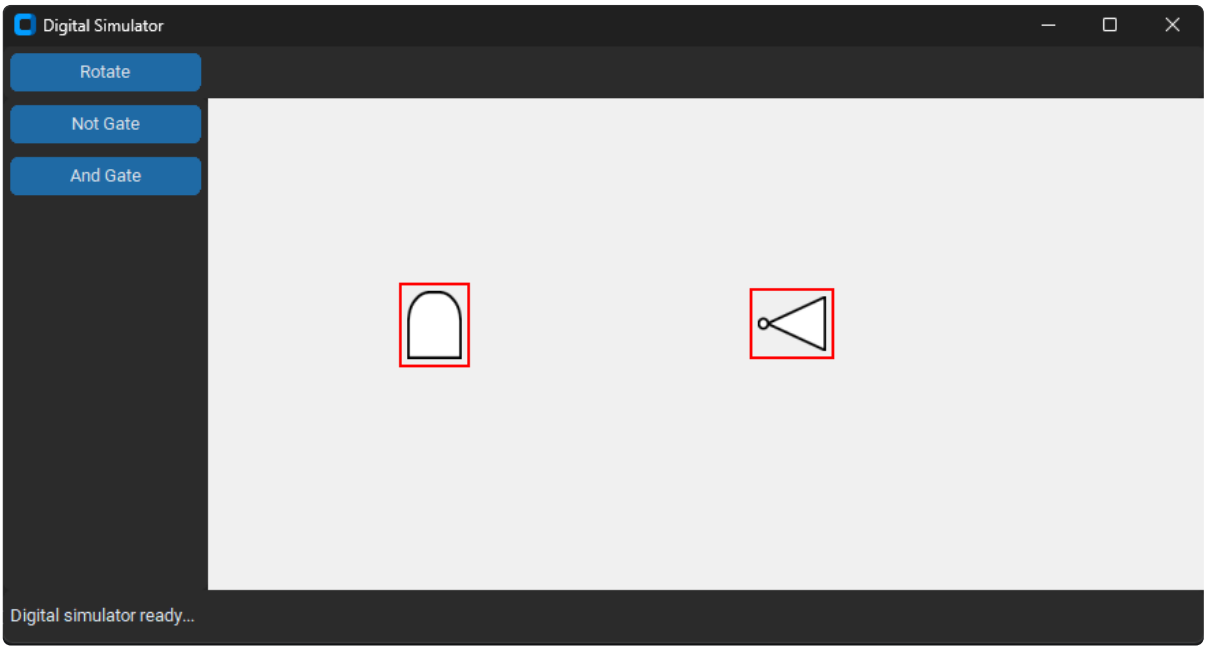
```
from pathlib import Path
from Comp_Lib.component import Comp
from Helper_Lib import Point
from Wire_Lib.connector import Connector


class AndGate(Comp):
    """And Gate Model"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/and_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
```

```
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        # self.move_connected_wires()
```

Component Rotation

Objectives:

- Rotate selected shape from top frame button
- Rotate selected shape using 'r' key on the keyboard
- Rotate shape in 90 deg increments



Comp_Lib/component.py

```
. . .

    def set_connector_visibility(self):
        """Set the connector visibility state"""
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

    def rotate(self):  # Added new method
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0
```

digital_simulator.py

```
. . .

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Rotate",
command=self.rotate_comp) # Modified to call the component rotation method
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

. . .

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add bindings here
        self.bind('<r>', self.rotate_comp)  # Added binding to the 'r' key

    def create_not_gate(self):
        self.gate1 = NotGate(self.canvas, 100, 100)
        self.canvas.comp_list.append(self.gate1)

. . .

    def rotate_comp(self, event=None):  # Added new component rotation method
        if self.canvas.mouse.selected_comp:
            self.canvas.mouse.selected_comp.rotate()
            self.canvas.redraw()


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

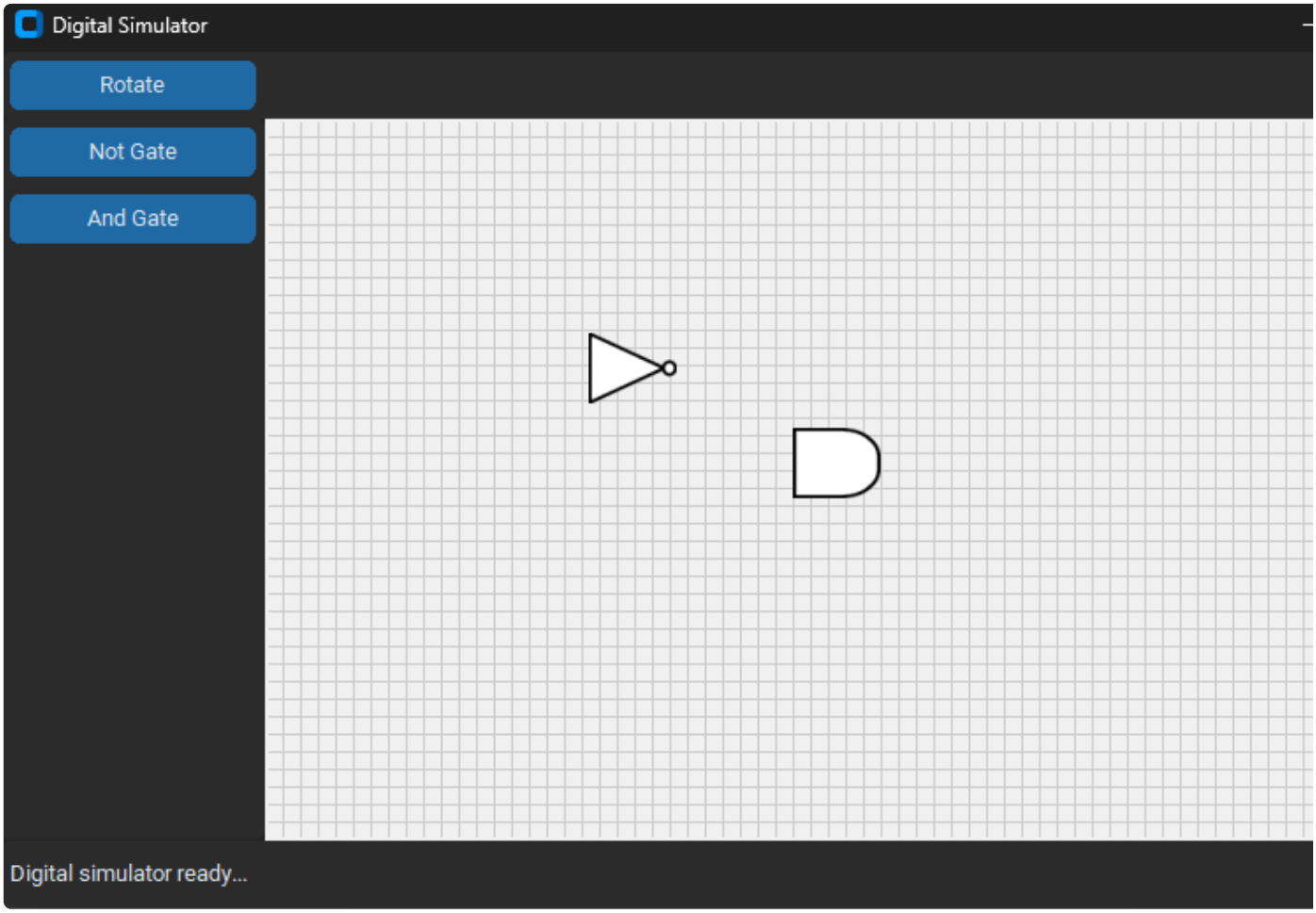
Note:

- Setting `event=None` in the `rotate_comp()` method allows it to be called as a button handler and a keyboard binding

Grid Class

Objectives:

- Draw a background grid on the canvas
- Add a snap-to-grid capability for all components



New Grid Class

UI_Lib/grid.py

```
class Grid:
    def __init__(self, canvas, grid_size):
        self.canvas = canvas
        self.grid_size = grid_size
        self.grid_visible = True

        self.grid_snap = self.grid_size
        self.draw()

    def draw(self):
```

```
        if self.grid_visible:
            w = self.canvas.winfo_width()  # Get current width of canvas
            h = self.canvas.winfo_height()  # Get current height of canvas

            # Creates all vertical lines at intervals of 100
            for i in range(0, w, self.grid_size):
                self.canvas.create_line([(i, 0), (i, h)],  fill='#cccccc',
tags='grid_line')

            # Creates all horizontal lines at intervals of 100
            for i in range(0, h, self.grid_size):
                self.canvas.create_line([(0, i), (w, i)],  fill='#cccccc',
tags='grid_line')

    def snap_to_grid(self, x, y):
        if self.grid_visible:
            x = round(x / self.grid_snap) * self.grid_snap
            y = round(y / self.grid_snap) * self.grid_snap
        return x, y
```

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid  # Added import for Grid class

class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

        self.grid = Grid(self, 10)  # Added grid object instantiation

    def redraw(self):
        self.delete('grid_line')  # Delete grid lines on each redraw
        self.grid.draw()  # Draw the grid
        self.tag_lower("grid_line")  # Lower the grid so it is in the
background
        for c in self.comp_list:
```

```
            c.update()
```

UI_Lib/mouse.py

```
from Helper_Lib import Point


class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_mouse_bind_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)

        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                x2, y2 = self.selected_comp.x2, self.selected_comp.y2
                self.offset1.set(x - x1, y - y1)
                self.offset2.set(x - x2, y - y2)
            else:
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                self.offset1.x, self.offset1.y =
self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)  # Added snap-to-
grid call
                self.offset1.set(x - x1, y - y1)
```

```
    def move_left_drag(self, event):
        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1 = event.x - self.offset1.x
                y1 = event.y - self.offset1.y
                x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)  # Added snap-
to-grid call
                x2 = event.x - self.offset2.x
                y2 = event.y - self.offset2.y
                x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)  # Added snap-
to-grid call
                self.selected_comp.x1, self.selected_comp.y1 = x1, y1
                self.selected_comp.x2, self.selected_comp.y2 = x2, y2
                self.canvas.redraw()
            else:
                x = event.x - self.offset1.x
                y = event.y - self.offset1.y
                x, y = self.canvas.grid.snap_to_grid(x, y)  # Added snap-to-
grid call
                self.selected_comp.x1, self.selected_comp.y1 = x, y
                self.canvas.redraw()

    def move_left_up(self, _event):
        if self.selected_comp:
            self.offset1.set(0, 0)
            self.offset2.set(0, 0)

    def select_hit_test(self, x, y):
        for s in self.canvas.comp_list:
            if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
                # print("Shape hit: ", s)
                self.selected_comp = s
                s.is_selected = True
                self.canvas.redraw()
                return

        # No shape hit - unselect all
        self.selected_comp = None
        self.unselect_all()

    def unselect_all(self):
        for s in self.canvas.comp_list:
            s.is_selected = False
        self.canvas.redraw()
```

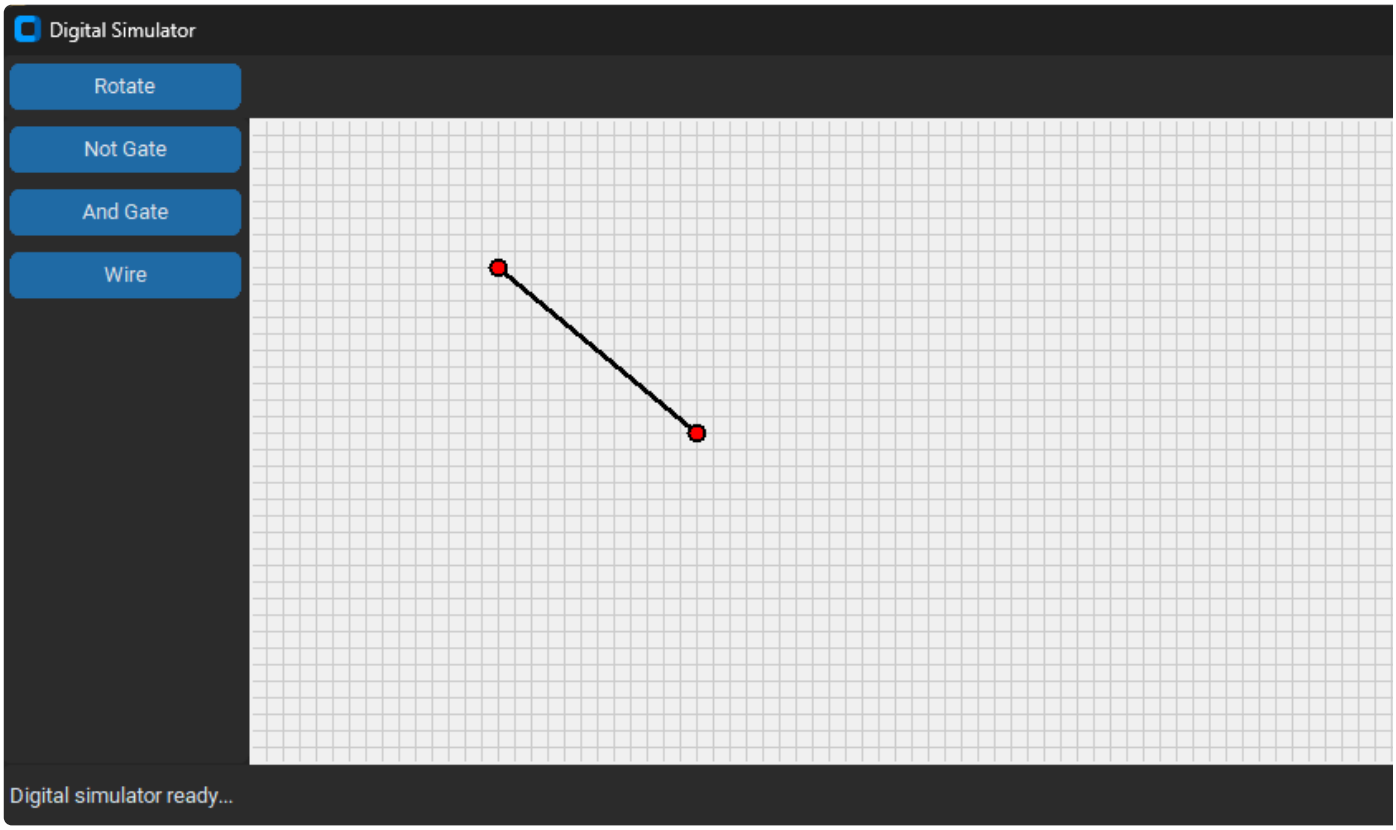
digital_simulator.py

```
. . .

        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)  # Add binding to
window resize event

. . .

    def on_window_resize(self, _event):  # Add method to redraw canvas whenever
window is resized
        self.canvas.redraw()


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

Draw Wires with Mouse

Objectives:

- Create wires using the mouse to "draw" the wire while dragging the mouse
- Create the wire in a "wire group" regardless if there is one wire or more than one wire in the group
- Add wire to the wire group is it intersects with an existing wire
- Show gate connectors while drawing wires
- Connect wires to gate connectors and resize connected wires if the gate is moved or rotated
- Wire selectors will be red ovals located at the two ends of the wire



New Wire Class

```
from Wire_Lib.wire_selector import WireSelector


class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3

        self.id = None
        self.state = False

        self.bbox = None
        self.is_selected = True
        self.sel_list = []
```

```
        self.selector = None

        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
width=self.border_width, tags='wire')

        self.s1_id, self.s2_id = None, None
        self.create_selectors()
        self.set_selector_visibility()

    def create_selectors(self):
        """Create selectors at the ends of the wire here"""
        self.s1_id = WireSelector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = WireSelector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_selectors()

    def update_position(self):
        """Update the position when the gate object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)  #
Update position

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_border_width(self):
        self.canvas.itemconfig(self.id, width=self.border_width)

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def set_selector_visibility(self):
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
```

```
                self.canvas.itemconfig(s.id, state='hidden')

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
            # self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2,
self.y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
            # self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1,
self.y1)

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Wire: " + " x1: " + str(self.x1) + " y1: " + str(self.y1) + \
            " x2: " + str(self.x2) + " y2: " + str(self.y2)
```

UI_Lib/mouse.py

```
from Helper_Lib import Point
from Wire_Lib import Connection  # Added import for connection class


class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None
        self.current_wire_obj = None

        self.start = Point(0, 0)
        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)
```

```
    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_mouse_bind_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def draw_wire_mouse_events(self):  # Added method to bind draw wire methods
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.draw_left_down)
        self.canvas.bind("<B1-Motion>", self.draw_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)

        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                x2, y2 = self.selected_comp.x2, self.selected_comp.y2
                self.offset1.set(x - x1, y - y1)
                self.offset2.set(x - x2, y - y2)
            else:
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                self.offset1.x, self.offset1.y =
self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
                self.offset1.set(x - x1, y - y1)

    def move_left_drag(self, event):
        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1 = event.x - self.offset1.x
                y1 = event.y - self.offset1.y
                x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
                x2 = event.x - self.offset2.x
                y2 = event.y - self.offset2.y
                x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
                self.selected_comp.x1, self.selected_comp.y1 = x1, y1
                self.selected_comp.x2, self.selected_comp.y2 = x2, y2
                self.canvas.redraw()
            else:
```

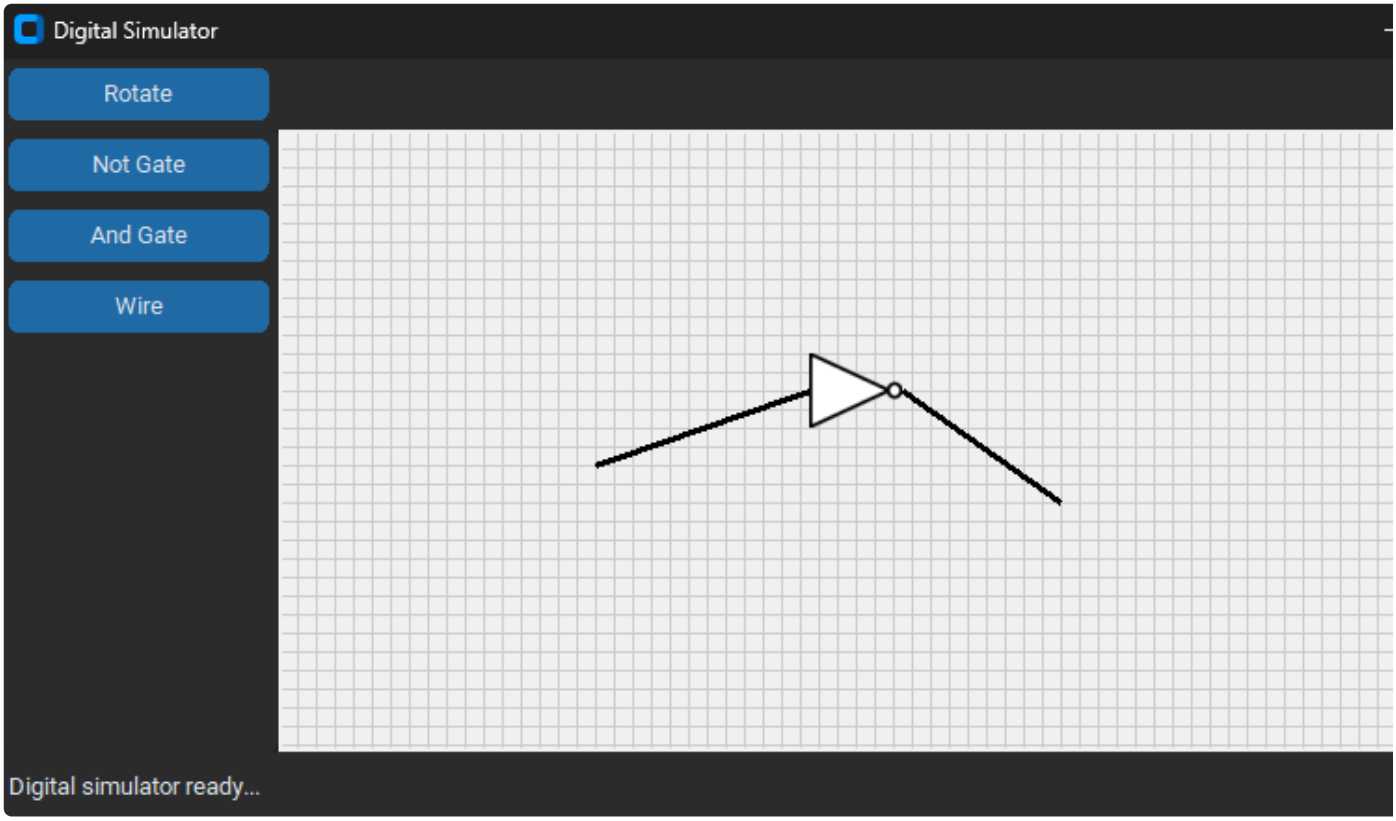
```
                x = event.x - self.offset1.x
                y = event.y - self.offset1.y
                x, y = self.canvas.grid.snap_to_grid(x, y)
                self.selected_comp.x1, self.selected_comp.y1 = x, y
                self.canvas.redraw()

    def move_left_up(self, _event):
        if self.selected_comp:
            self.offset1.set(0, 0)
            self.offset2.set(0, 0)

    def draw_left_down(self, event):  # Added method for draw left down
        if self.current_wire_obj:
            self.unselect_all()
            self.start.x = event.x
            self.start.y = event.y
            self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

            self.current_wire_obj.x1, self.current_wire_obj.y1 = self.start.x,
self.start.y
            self.current_wire_obj.x2, self.current_wire_obj.y2 = self.start.x,
self.start.y

            if self.current_wire_obj is not None:
                self.select_connector(self.current_wire_obj, "begin",
self.start.x, self.start.y)

    def draw_left_drag(self, event):  # Added method for draw left drag
        if self.current_wire_obj:
            shape = self.current_wire_obj
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            shape.x1, shape.y1 = self.start.x, self.start.y
            shape.x2, shape.y2 = x, y
            self.canvas.redraw()

    def draw_left_up(self, event):  # Added method for draw left up
        self.select_connector(self.current_wire_obj, "end", event.x, event.y)
        self.canvas.hide_connectors()
        # self.move_mouse_bind_events()

    def select_hit_test(self, x, y):
        for s in self.canvas.comp_list:
            if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
                # print("Shape hit: ", s)
```

```
                self.selected_comp = s
                s.is_selected = True
                self.canvas.redraw()
                return

        # No shape hit - unselect all
        self.selected_comp = None
        self.unselect_all()

    def unselect_all(self):
        for s in self.canvas.comp_list:
            s.is_selected = False
        self.canvas.redraw()

    def select_connector(self, wire_obj, wire_end, x, y):  # Added method to
see if line end hits a gate connector
        for comp in self.canvas.comp_list:
            if not self.canvas.gettags(comp.id)[0] == 'wire':
                conn = comp.check_connector_hit(x, y)
                if conn:
                    if wire_end == "begin":
                        wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                    elif wire_end == "end":
                        wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, self.current_wire_obj, wire_end)
                    comp.wire_list.append(a_conn)
                    self.canvas.redraw()
```

digital_simulator.py

```
. . .

    def create_wire(self):  # updated method
        self.wire = Wire(self.canvas, 0, 0, 0, 0)
        self.canvas.comp_list.append(self.wire)
        self.canvas.mouse.current_wire_obj = self.wire
        self.canvas.mouse.draw_wire_mouse_events()

. . .
```

Move Wire if connected Gate moves



Comp_Lib/not_gate.py

```
. . .

    def update_connectors(self):

                . . .

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()  # Added call to move connected wires
```

Add the same call at end of the `update_connectors()` method in the And Gate Class

Comp_Lib/component.py

```
. . .

    def check_connector_hit(self, x, y):  # Added new method
        """Hit test to see if a connector is at the provided x, y
coordinates"""
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_wires(self):  # Added new method
        """Resize connected wires if the shape is moved"""
        for connection in self.wire_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.wire_end == "begin":
                        connection.wire_obj.x1 = connector.x
                        connection.wire_obj.y1 = connector.y
                    elif connection.wire_end == "end":
                        connection.wire_obj.x2 = connector.x
                        connection.wire_obj.y2 = connector.y
```

Other Gates, Switch, LED, and Text Classes

Objectives:

- Create Nand Gate class
- Create Nor Gate class
- Create Xor Gate class
- Create Xnor Gate class
- Create Switch class
- Create LED class
  - Make LED color configurable
  - Make LED size configurable
- Create Text class

UI_Lib/nand_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class NandGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/gates/nand_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
```

```
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "in1":
                self.in1_state = wire.wire_obj.state
                self.out_state = not(self.in1_state and self.in2_state)
            elif wire.connector_obj.name == "in2":
                self.in2_state = wire.wire_obj.state
                self.out_state = not(self.in1_state and self.in2_state)
            elif wire.connector_obj.name == "out":
                wire.wire_obj.state = self.out_state
```

UI_Lib/nor_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class NorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/gates/nor_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
```

```
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "in1":
                self.in1_state = wire.wire_obj.state
                self.out_state = not(self.in1_state or self.in2_state)
            elif wire.connector_obj.name == "in2":
                self.in2_state = wire.wire_obj.state
                self.out_state = not(self.in1_state or self.in2_state)
            elif wire.connector_obj.name == "out":
```

```
                wire.wire_obj.state = self.out_state
```

UI_Lib/xor_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class XorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/gates/xor_60x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
```

```
        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "in1":
                self.in1_state = wire.wire_obj.state
                self.out_state = self.in1_state != self.in2_state
            elif wire.connector_obj.name == "in2":
```

```
                self.in2_state = wire.wire_obj.state
                self.out_state = self.in1_state != self.in2_state
            elif wire.connector_obj.name == "out":
                wire.wire_obj.state = self.out_state
```

UI_Lib/xnor_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class XnorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/gates/xnor_60x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
```

```
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
```

```
            if wire.connector_obj.name == "in1":
                self.in1_state = wire.wire_obj.state
                self.out_state = not(self.in1_state != self.in2_state)
            elif wire.connector_obj.name == "in2":
                self.in2_state = wire.wire_obj.state
                self.out_state = not(self.in1_state != self.in2_state)
            elif wire.connector_obj.name == "out":
                wire.wire_obj.state = self.out_state
```

UI_Lib/switch.py

```
from pathlib import Path
from PIL import Image

from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector
from Helper_Lib import Point


class Switch(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "switch"
        self.out_state = False  # OFF state
        self.subckt_conn = None

        self.filename_sw_on = Path(__file__).parent /
"../images/switch/switch_on.png"
        self.on_image = Image.open(self.filename_sw_on)
        self.filename_sw_off = Path(__file__).parent /
"../images/switch/switch_off.png"
        self.off_image = Image.open(self.filename_sw_off)
        self.filename = self.filename_sw_off

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.out1_id = None
        self.create_connectors()
        self.set_connector_visibility()
```

```
    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def toggle_switch(self):
        if self.filename == self.filename_sw_off:
            self.filename = self.filename_sw_on
        else:
            self.filename = self.filename_sw_off

    def create_connectors(self):  # Added new method
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 1 connector: out1
        self.out1_id = Connector(self.canvas, "out", center.x + w / 2,
center.y)
        self.conn_list = [self.out1_id]
        self.set_connector_visibility()

    def update_connectors(self):  # Added new method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Update connector position based on angle
        if self.angle == 0:
            self.out1_id.x, self.out1_id.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.out1_id.x, self.out1_id.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.out1_id.x, self.out1_id.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.out1_id.x, self.out1_id.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()
```

```
        self.move_connected_wires()

    def toggle_state(self):
        self.out_state = not self.out_state
        self.toggle_switch()

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
            self.wire_list[0].wire_obj.set_wire_state()
```

UI_Lib/led.py

```
from pathlib import Path

from Wire_Lib.connector import Connector
from Comp_Lib.component import Comp
from Helper_Lib.point import Point


class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False
        self.angle = 180

        self.led_color = self.canvas.led_color  # red, green, yellow, blue
        self.led_size = self.canvas.led_size  # small, large

        if self.led_size == "large":
            self.w = 40
            self.h = 40
            color_led_str = "../images/led/led_on_" + self.led_color +
"_large.png"
            self.filename_led_on = Path(__file__).parent / color_led_str
            self.filename_led_off = Path(__file__).parent /
"../images/led/led_off_large.png"
        elif self.led_size == "small":
            self.w = 20
            self.h = 20
            color_led_str = "../images/led/led_on_" + self.led_color +
"_small.png"
```

```
            self.filename_led_on = Path(__file__).parent / color_led_str
            self.filename_led_off = Path(__file__).parent /
"../images/led/led_off_small.png"
        self.filename = self.filename_led_off

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.in1_id = None
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_state()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def update_state(self):
        if self.state:
            self.filename = self.filename_led_on
        else:
            self.filename = self.filename_led_off

    def create_connectors(self):  # Added new method
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 1 connector: out
        self.in1_id = Connector(self.canvas, "in", center.x - w / 2, center.y)
        self.conn_list = [self.in1_id]
        self.set_connector_visibility()

    def update_connectors(self):  # Added new method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
```

```
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Update connector position based on angle
        if self.angle == 0:
            self.in1_id.x, self.in1_id.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.in1_id.x, self.in1_id.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.in1_id.x, self.in1_id.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.in1_id.x, self.in1_id.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        if self.wire_list:
            self.state = self.wire_list[0].wire_obj.state
```

UI_Lib/text.py

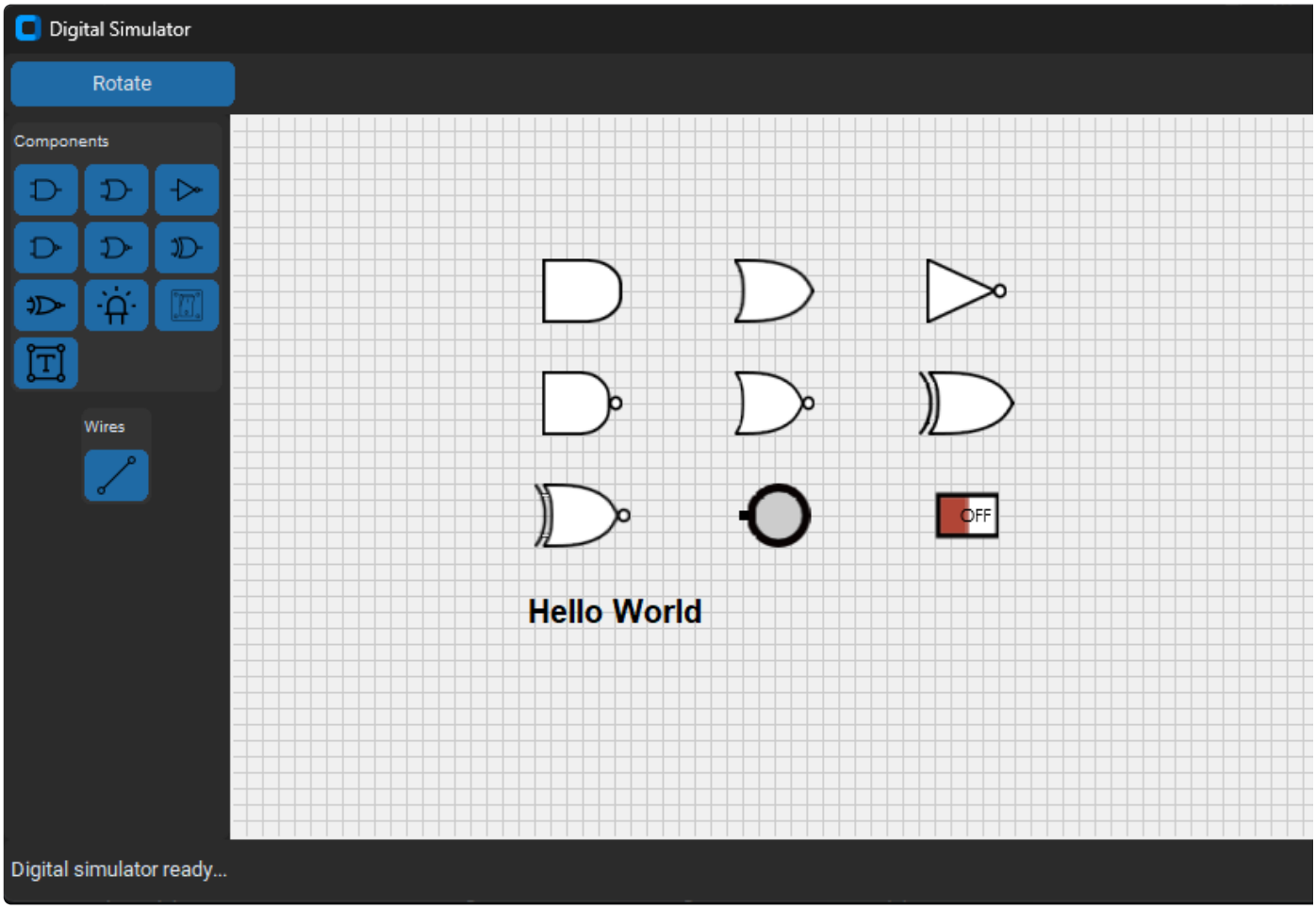
```
from Comp_Lib.component import Comp


class TextShape(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "text_shape"
        self.text = 'Hello World'

        self.fill_color = "black"
        self.border_color = "black"
        self.border_width = 3

        self.id = self.canvas.create_text(self.x1, self.y1,
                                  text=self.text, fill=self.fill_color,
                                  font='Helvetica 15 bold',
                                  angle=self.angle, tags="text")

        self.update_bbox()
```

```
        self.create_selector()

    def update(self):
        self.update_position()
        self.update_rotation()
        self.update_bbox()
        self.update_selector()

    def update_rotation(self):
        self.canvas.itemconfig(self.id, angle=self.angle)
```

Left Frame Class

Objectives:

- Create a custom left frame class
- Component button frame with picture buttons for components in a grid
- Wire button frame with picture buttons for wires in a grid



digital_simulator.py

```
import customtkinter as ctk
from UI_Lib import Canvas, LeftFrame  # Added import for left frame class

ctk.set_appearance_mode("System")  # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue")  # Themes: "blue" (standard), "green",
"dark-blue"


class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100")  # w, h, x, y
        self.title("Digital Simulator")

        self.canvas = Canvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = LeftFrame(self, self.canvas)  # Modified to use the
left frame class
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Rotate",
command=self.rotate_comp)  # Removed left frame buttons
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)

    def rotate_comp(self, _event=None):
        if self.canvas.mouse.selected_comp:
```

```
            self.canvas.mouse.selected_comp.rotate()
            self.canvas.redraw()

    def on_window_resize(self, _event):
        self.canvas.redraw()


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

UI_Lib/left_frame.py - new class

```
import customtkinter as ctk
from UI_Lib.comp_button_frame import CompButtonFrame
from UI_Lib.wire_button_frame import WireButtonFrame


class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.comp_button_frame = CompButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.wire_button_frame = WireButtonFrame(self, self.canvas)
        self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

UI_Lib/comp_button_frame.py

```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import AndGate, OrGate, NotGate, NandGate, NorGate, XorGate,
XnorGate
from Comp_Lib import LED, Switch, Text
```

```
class CompButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("and_gate", "../icons/and.png"),
                            ("or_gate", "../icons/or.png"),
                            ("not_gate", "../icons/not.png"),
                            ("nand_gate", "../icons/nand.png"),
                            ("nor_gate", "../icons/nor.png"),
                            ("xor_gate", "../icons/xor.png"),
                            ("xnor_gate", "../icons/xnor.png"),
                            ("led", "../icons/led.png"),
                            ("switch", "../icons/switch.png"),
                            ("text", "../icons/text.png")]

        self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Components", font=
("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:
            a_image = ctk.CTkImage(light_image=Image.open
                                   (Path(__file__).parent / button[1]),
                                   dark_image=Image.open
                                   (Path(__file__).parent / button[1]),
                                   size=(24, 24))
            self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                                           command=lambda
a_name=button[0]:self.create_events(a_name))
            self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
            ToolTip(self.button_id, msg=button[0])
            row_num, col_num = self.update_grid_numbers(row_num, col_num)

    def create_events(self, name):
        gate = None
```

```
        if name == "and_gate":
            gate = AndGate(self.canvas, 100, 100)
        elif name == "or_gate":
            gate = OrGate(self.canvas, 100, 100)
        elif name == "not_gate":
            gate = NotGate(self.canvas, 100, 100)
        elif name == "nand_gate":
            gate = NandGate(self.canvas, 100, 100)
        elif name == "nor_gate":
            gate = NorGate(self.canvas, 100, 100)
        elif name == "xor_gate":
            gate = XorGate(self.canvas, 100, 100)
        elif name == "xnor_gate":
            gate = XnorGate(self.canvas, 100, 100)
        elif name == "led":
            gate = LED(self.canvas, 100, 100)
        elif name == "switch":
            gate = Switch(self.canvas, 100, 100)
        elif name == "text":
            gate = Text(self.canvas, 100, 100)
        self.canvas.comp_list.append(gate)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

    @staticmethod
    def update_grid_numbers(row, column):
        column += 1
        if column > 2:
            column = 0
            row += 1
        return row, column
```

UI_Lib/wire_button_frame.py

```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import Wire


class WireButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
```

```
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.wire = None

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Wires", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        wire_image = ctk.CTkImage(light_image=Image.open
                                  (Path(__file__).parent /
"../icons/straight_line.png"),
                                  dark_image=Image.open
                                  (Path(__file__).parent /
"../icons/straight_line.png"),
                                  size=(24, 24))

        wire1_button = ctk.CTkButton(self, text="", image=wire_image, width=30,
                                     command=self.create_wire)
        wire1_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

    # Shape button handlers
    def create_wire(self):
        wire = Wire(self.canvas, 0, 0, 0, 0)
        self.canvas.comp_list.append(wire)
        self.canvas.mouse.current_wire_obj = wire
        self.canvas.show_connectors()
        self.canvas.mouse.draw_wire_mouse_events()
```

UI_Lib/canvas.py

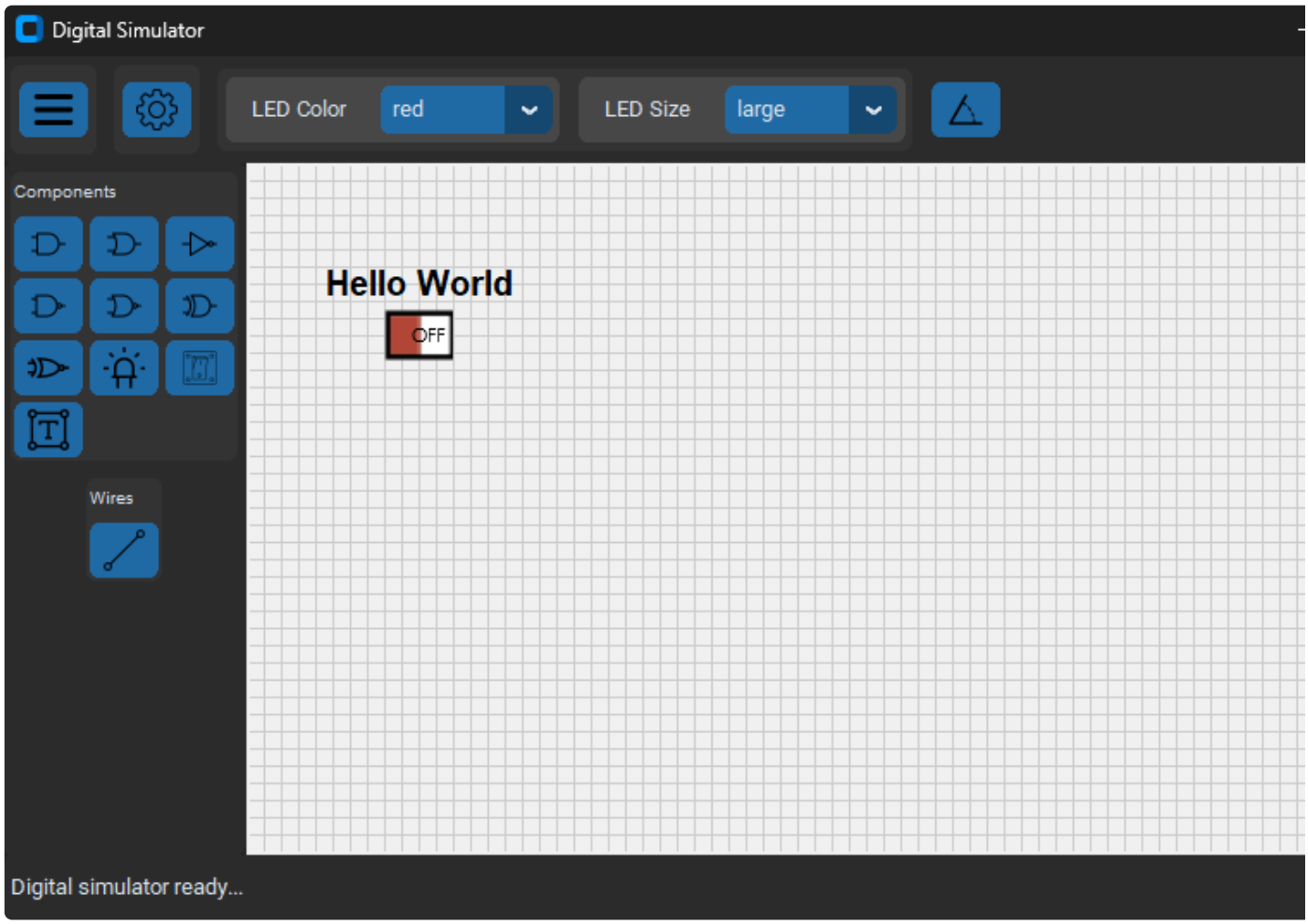
```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)
```

```
        self.led_color = "red"
        self.led_size = "large"
        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

        self.grid = Grid(self, 10)

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

    def show_connectors(self):
        for s in self.comp_list:
            s.is_drawing = True
        self.redraw()

    def hide_connectors(self):
        for s in self.comp_list:
            s.is_drawing = False
        self.redraw()
```

Top Frame Class

Objectives:

- Create a custom Top Frame class
- Add file menu control
- Add settings control
- Add LED color and size control
- Add rotation button
- Add help menu control



digital_simulator.py

```
import customtkinter as ctk
from UI_Lib import Canvas, LeftFrame, TopFrame  # added import for top frame
class

ctk.set_appearance_mode("System")  # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue")  # Themes: "blue" (standard), "green",
"dark-blue"


class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100")  # w, h, x, y
        self.title("Digital Simulator")
```

```
        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas)  # Modified to use the top
frame class
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")  # Removed top frame buttons
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)

    def rotate_comp(self, _event=None):
        if self.canvas.mouse.selected_comp:
            self.canvas.mouse.selected_comp.rotate()
            self.canvas.redraw()

    def on_window_resize(self, _event):
        self.canvas.redraw()


if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

UI_Lib/top_frame.py - new class

```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from UI_Lib.file_menu_frame import FileMenuFrame
from UI_Lib.settings_frame import SettingsFrame
```

```
from UI_Lib.led_frame import LedFrame
from UI_Lib.help_frame import HelpFrame


class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add Top Frame widget here
        file_frame = FileMenuFrame(self.parent, self, self.canvas)
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        settings_frame = SettingsFrame(self.parent, self, self.canvas)
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        led_frame = LedFrame(self.parent, self, self.canvas)
        led_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        help_frame = HelpFrame(self.parent, self, self.canvas)
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)

        a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               dark_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               size=(24, 24))
        self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.rotate_comp)
        self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
        ToolTip(self.button_id, msg="Rotate selected component")

    def rotate_comp(self):
        self.parent.rotate_comp(_event=None)
```

UI_Lib/file_menu_frame.py - new class

```
import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json
from PIL import Image
```

```
from Comp_Lib import AndGate, OrGate, NandGate, NorGate, XorGate, XnorGate,
Switch, LED, Text
from Wire_Lib import Wire


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'and': AndGate,
                              'nand': NandGate,
                              'or': OrGate,
                              'nor': NorGate,
                              'xor': XorGate,
                              'xnor': XnorGate,
                              'switch': Switch,
                              'wire': Wire,
                              'led': LED,
                              'text': Text}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
command=self.new_diagram)
        new_btn.pack(pady=5)

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                dark_image=Image.open
```

```
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.canvas.delete("all")
        self.canvas.comp_list = []

    def load_diagram(self):
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                obj_dict = json.load(file)
                for obj_type, attributes in obj_dict.items():
                    if obj_type.split()[0] == "wire":
                        obj = self.obj_type_dict[obj_type.split()[0]]
(self.parent.parent.active_canvas,

attributes[0], attributes[1],

attributes[2], attributes[3])
                    else:
                        obj = self.obj_type_dict[obj_type.split()[0]]
(self.parent.parent.active_canvas,

attributes[0], attributes[1])
                    obj.angle = attributes[3]
                    self.parent.parent.active_canvas.comp_list.append(obj)
                self.canvas.redraw()

        except FileNotFoundError:
            with open('untitled.canvas', 'w') as _file:
                pass
            self.canvas.comp_list = []

    def save_diagram(self):
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            obj_dict = {f'{obj.type} {id}': (obj.x1, obj.y1, obj.x2, obj.y2,
obj.angle) for
```

```
                        id, obj in
enumerate(self.parent.parent.active_canvas.comp_list)}
            json.dump(obj_dict, file)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False
```

UI_Lib/settings_frame.py - new class

```
import customtkinter as ctk
from PIL import Image


class SettingsFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        def grid_switch_event():
            if canvas.grid.grid_visible:
                self.canvas.grid.grid_visible = False
            else:
                self.canvas.grid.grid_visible = True
            self.canvas.redraw()

        switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
command=grid_switch_event,
                                       variable=switch_var, onvalue="on",
offvalue="off")
        switch.pack(padx=5, pady=5)
```

```
        grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=
("Helvetica", 10), height=20)
        grid_size_label.pack(padx=5, pady=5, anchor="w")

        def optionmenu_callback(choice):
            self.canvas.grid.grid_size = int(choice)
            self.canvas.redraw()

        optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=["5", "10",
"20", "30", "40", "50"],
                                                command=optionmenu_callback)
        optionmenu.pack(padx=5, pady=5)
        optionmenu.set("10")

        grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=
("Helvetica", 10), height=20)
        grid_snap_label.pack(padx=5, pady=5, anchor="w")

        def snap_option_callback(choice):
            if choice == "Grid Size":
                self.canvas.grid.grid_snap = canvas.grid.grid_size
            else:
                self.canvas.grid.grid_snap = int(choice)
            canvas.redraw()

        snap_option = ctk.CTkOptionMenu(self.menu_frame, values=["Grid Size",
"5", "10", "20", "30", "40", "50"],
                                                command=snap_option_callback)
        snap_option.pack(padx=5, pady=5)
        snap_option.set("Grid Size")

        self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,
text="Appearance Mode:", anchor="w")
        self.appearance_mode_label.pack(padx=5, pady=5)
        self.appearance_mode_optionemenu = ctk.CTkOptionMenu(self.menu_frame,
                                                             values=
["Light", "Dark", "System"],

command=self.change_appearance_mode_event)
        self.appearance_mode_optionemenu.pack(padx=5, pady=5)
        self.appearance_mode_optionemenu.set("Dark")

        my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
                                dark_image=Image.open
```

```
("D:/EETools/DiagramEditor/icons/settings.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    @staticmethod
    def change_appearance_mode_event(new_appearance_mode: str):
        ctk.set_appearance_mode(new_appearance_mode)
```

UI_Lib/led_frame.py - new class

```
import customtkinter as ctk
from tktooltip import ToolTip


class LedFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.led_color = "red"  # red, yellow, blue, green
        self.lec_size = "large"  # large, small

        self.init_led_color_control(self)
        self.init_led_size_control(self)

    def init_led_color_control(self, parent_frame):
        move_frame = ctk.CTkFrame(parent_frame, width=150)
        move_frame.configure(fg_color=("gray28", "gray28"))  # set frame color
        move_frame.pack(side=ctk.LEFT, padx=5, pady=5)
```

```
        image_label = ctk.CTkLabel(move_frame, text="LED Color",
corner_radius=10)
        image_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            self.canvas.led_color = choice

        option_menu = ctk.CTkOptionMenu(move_frame, values=["red", "yellow",
"blue", "green"], width=100,
                                        command=option_menu_callback)
        option_menu.pack(side=ctk.LEFT, padx=5, pady=5)
        option_menu.set("red")

        ToolTip(option_menu, msg="Set LED color")

    def init_led_size_control(self, parent_frame):
        move_frame = ctk.CTkFrame(parent_frame, width=150)
        move_frame.configure(fg_color=("gray28", "gray28"))  # set frame color
        move_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        image_label = ctk.CTkLabel(move_frame, text="LED Size",
corner_radius=10)
        image_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            self.canvas.led_size = choice

        option_menu = ctk.CTkOptionMenu(move_frame, values=["large", "small"],
width=100,
                                        command=option_menu_callback)
        option_menu.pack(side=ctk.LEFT, padx=5, pady=5)
        option_menu.set("large")

        ToolTip(option_menu, msg="Set LED size")
```

UI_Lib/help_frame.py - new class

```
import customtkinter as ctk
from tkinter import messagebox
from PIL import Image
```

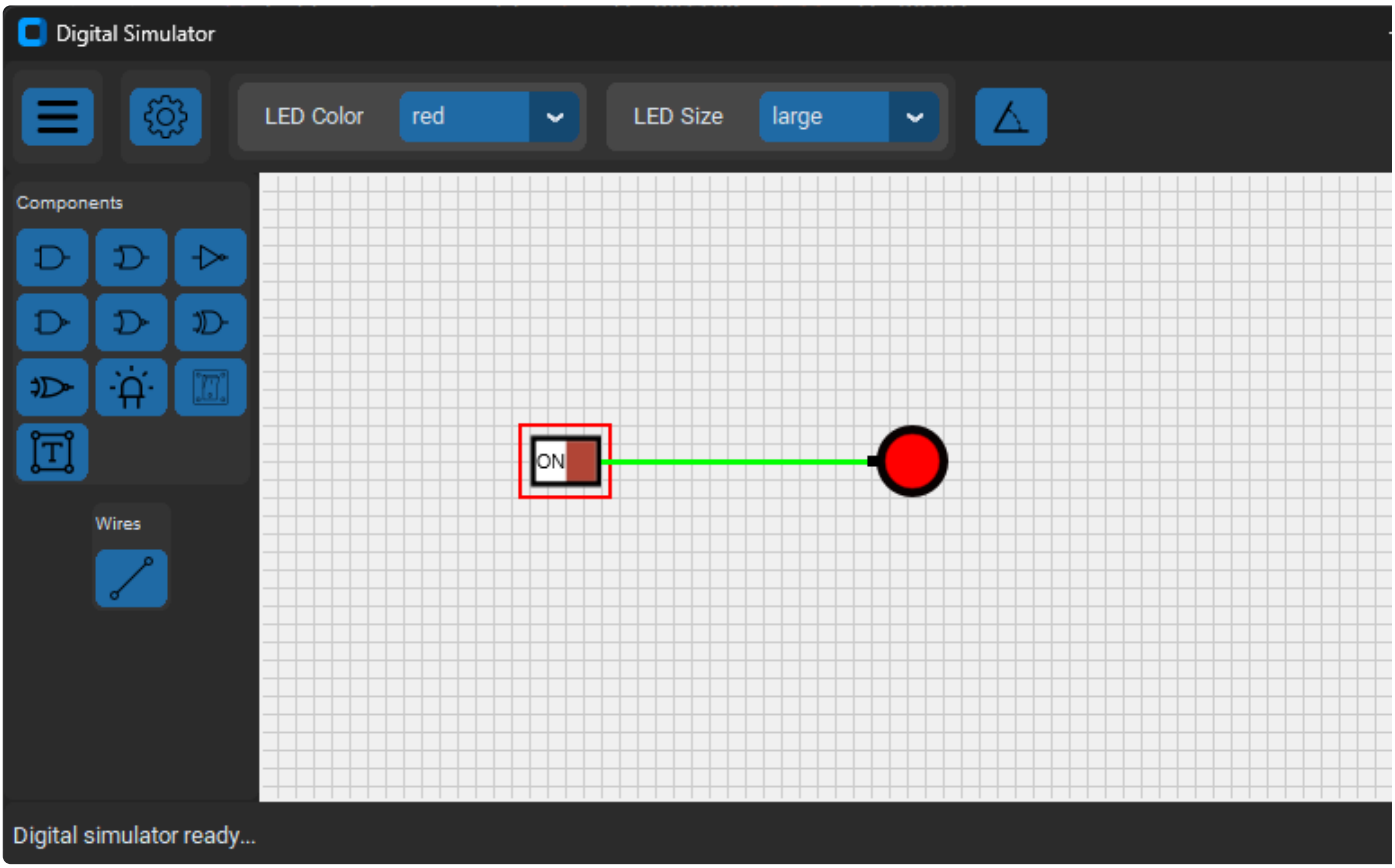
```
class HelpFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.window = window
        self.parent = parent
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        about_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
                                   dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
                                   size=(24, 24))

        about_button = ctk.CTkButton(self.menu_frame, text="About",
image=about_image, width=30,
                                     command=self.show_about_dialog)
        about_button.pack(side=ctk.TOP,padx=5, pady=5)

        my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
                                   dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
                                   size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            menu_pos_x = self.canvas.winfo_width()
            self.menu_frame.place(x=menu_pos_x + 50, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
```

```
            self.menu_on = False

    @staticmethod
    def show_about_dialog():
        messagebox.showinfo("About Digital Simulator", "Digital Simulator
v0.1\n" +
                            "Author: Rick A. Crist\n" + "2023")
```

Switch-LED Circuit Logic

Objectives:

- Create a circuit diagram with a switch, wire, and LED
- Toggle the switch with the right mouse button
- Propagate the logic signal from the switch to the wire to the LED



digital_simulator.py

```
. . .
```

```
        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)
        self.canvas.bind('<Button-3>', self.canvas.edit_shape)  # added new
binding to right mouse button

. . .
```

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Comp_Lib import Switch, Text  # Added new import for swtich and text
classes

. . .

    def edit_shape(self, _event=None):  # Added new method
        if isinstance(self.mouse.selected_comp, Switch):
            self.mouse.selected_comp.toggle_state()
            self.redraw()
        elif isinstance(self.mouse.selected_comp, Text):
            pass
```

Comp_Lib/switch.py

```
. . .

    def toggle_state(self):  # Method to toggle the output state
        self.out_state = not self.out_state
        self.toggle_switch()

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
```

Wire_Lib/wire.py

```
from Wire_Lib.wire_selector import WireSelector


class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3

        self.id = None
        self.state = False  # Variable to store the wire logic state

 . . .

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_wire_color()  # Added call to update the wire color
        self.update_selectors()
        self.set_selector_visibility()

. . .

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_wire_color(self):  # Method to update wire color based on logic
state
        if self.state:
            self.canvas.itemconfig(self.id, fill="#00ff00")
        else:
            self.canvas.itemconfig(self.id, fill=self.fill_color)

. . .
```

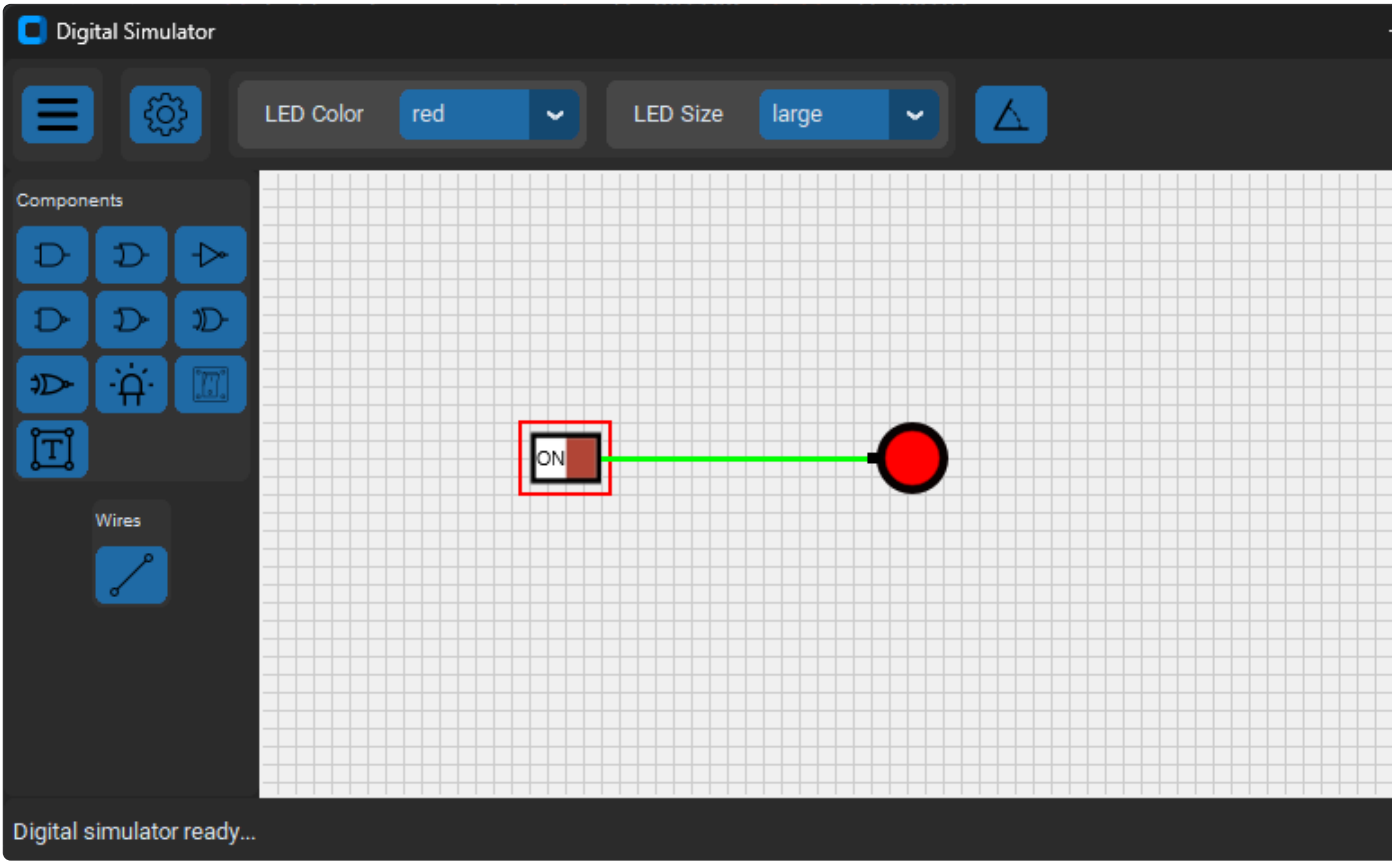
Comp_Lib/led.py

```
. . .

class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False  # Variable to store the led logic state
        self.angle = 180

. . .

    def update(self):
        self.set_logic_level()  # Call set logic level method to update the led
logic state
        self.update_led_color()  # Call to update state
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()


    def update_led_color(self):  # Method to update led color based on the led
state
        if self.state:
            self.filename = self.filename_led_on
        else:
            self.filename = self.filename_led_off

. . .

       self.move_connected_wires()

    def set_logic_level(self):  # Method to set the led state based on
connected wire state
        if self.wire_list:
            self.state = self.wire_list[0].wire_obj.state
```

JSON Encoder & Decoder

Objectives:

- Create a circuit diagram with a switch, wire, and LED
- Toggle the switch with the right mouse button
- Propagate the logic signal from the switch to the wire to the LED



digital_simulator.py

```
. . .

        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)
        self.canvas.bind('<Button-3>', self.canvas.edit_shape)  # added new
binding to right mouse button

. . .
```

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Comp_Lib import Switch, Text  # Added new import for swtich and text
classes

. . .

    def edit_shape(self, _event=None):  # Added new method
        if isinstance(self.mouse.selected_comp, Switch):
            self.mouse.selected_comp.toggle_state()
            self.redraw()
        elif isinstance(self.mouse.selected_comp, Text):
            pass
```

Comp_Lib/switch.py

```
. . .

    def toggle_state(self):  # Method to toggle the output state
        self.out_state = not self.out_state
        self.toggle_switch()

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
```

Wire_Lib/wire.py

```
from Wire_Lib.wire_selector import WireSelector


class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

```
        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3

        self.id = None
        self.state = False  # Variable to store the wire logic state

 . . .

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_wire_color()  # Added call to update the wire color
        self.update_selectors()
        self.set_selector_visibility()

. . .

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_wire_color(self):  # Method to update wire color based on logic
state
        if self.state:
            self.canvas.itemconfig(self.id, fill="#00ff00")
        else:
            self.canvas.itemconfig(self.id, fill=self.fill_color)

. . .
```

Comp_Lib/led.py

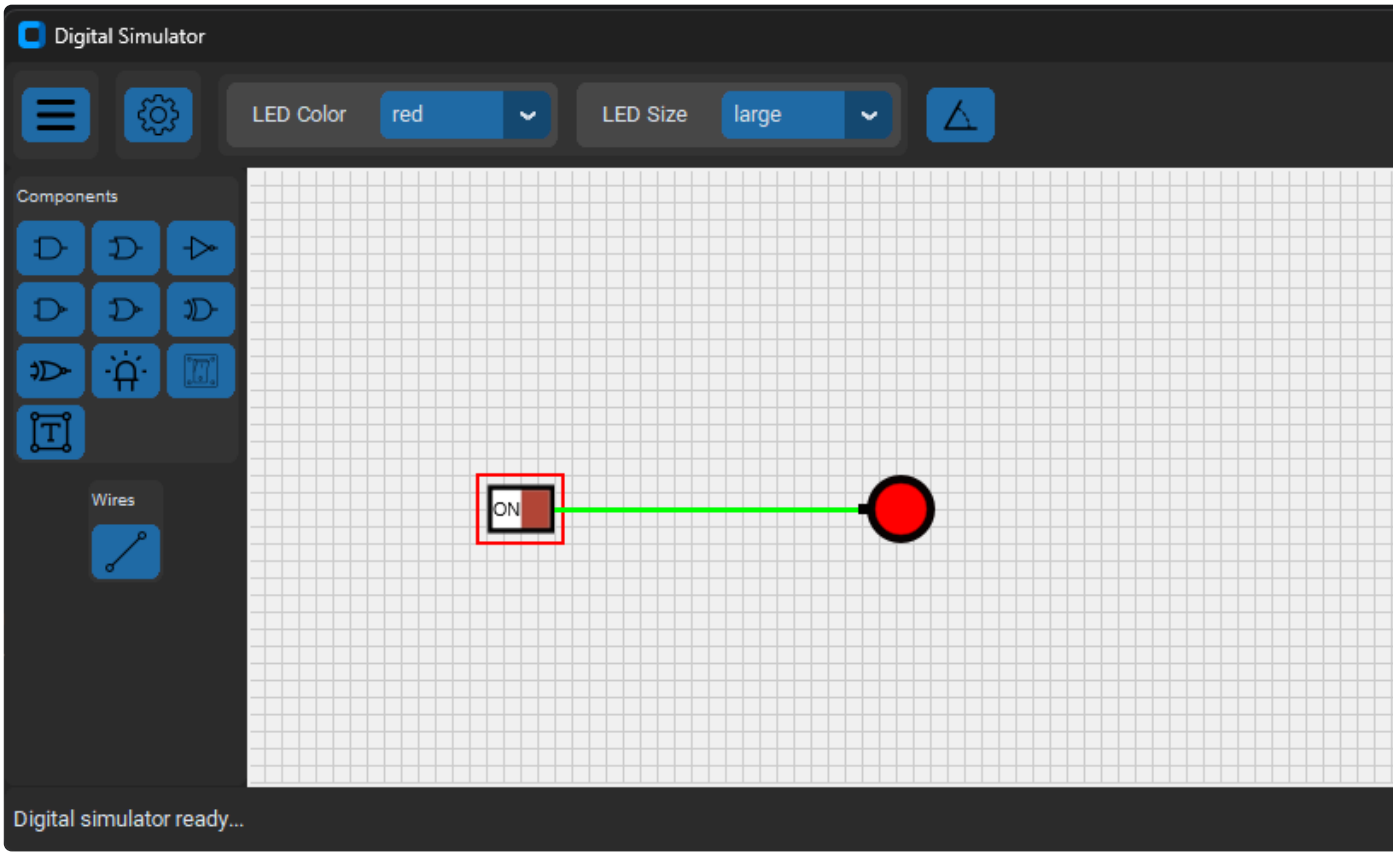
```
. . .

class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False  # Variable to store the led logic state
        self.angle = 180

. . .
```

```
    def update(self):
        self.set_logic_level()  # Call set logic level method to update the led
logic state
        self.update_led_color()  # Call to update state
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()


    def update_led_color(self):  # Method to update led color based on the led
state
        if self.state:
            self.filename = self.filename_led_on
        else:
            self.filename = self.filename_led_off

. . .

       self.move_connected_wires()

    def set_logic_level(self):  # Method to set the led state based on
connected wire state
        if self.wire_list:
            self.state = self.wire_list[0].wire_obj.state
```

File Save & Load

Objectives:

- Use custom JSON encoder class to save circuit in a .json file
- Use custom JSON decoder class to load saved circuit
- Verify that the loaded circuit is operational



UI_Lib/file_menu_frame.py

```
import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json  # import json library
from PIL import Image

from Comp_Lib import AndGate, OrGate, NandGate, NorGate, XorGate, XnorGate,
Switch, LED, Text
from Wire_Lib import Wire, Connection  # added import for connection class


class MyEncoder(json.JSONEncoder):  # Added custom JSON encoder class
    def default(self, o):
        if hasattr(o, "reprJson"):
            return o.reprJson()
        else:
            return super().default(o)


class JSONDCoder(json.JSONDecoder):  # Added custom JSON decoder class
    def __init__(self):
```

```
        json.JSONDecoder.__init__(self, object_hook=JSONDCoder.from_dict)

    @staticmethod
    def from_dict(_d):
        return _d


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'and': AndGate,
                              'nand': NandGate,
                              'or': OrGate,
                              'nor': NorGate,
                              'xor': XorGate,
                              'xnor': XnorGate,
                              'switch': Switch,
                              'wire': Wire,
                              'led': LED,
                              'text': Text}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
command=self.new_diagram)
        new_btn.pack(pady=5)

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
```

```
                                dark_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.canvas.delete("all")
        self.canvas.comp_list = []

    def load_diagram(self):  # Modified to load json file
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                d = json.load(file)
                print(d)

                                # Add components to canvas comp list
                for item in d:
                    if item['type'] == "wire":
                        wire = Wire(self.canvas, x1=item['x1'], y1=item['y1'],
x2=item['x2'], y2=item['y2'])
                        self.canvas.comp_list.append(wire)
                    elif item['type'] == "switch":
                        switch = Switch(self.canvas, x1=item['x1'],
y1=item['y1'])
                        switch.angle = item['angle']
                        self.canvas.comp_list.append(switch)
                    elif item['type'] == "led":
                        led = LED(self.canvas, x1=item['x1'], y1=item['y1'])
                        led.angle = item['angle']
                        self.canvas.comp_list.append(led)

                # Add connections to comp wire lists
                for item in d:
                    if item['type'] == "switch":
                        wire_list = item['wire_list']
                        for wire_item in wire_list:
                            x1 = wire_item['wire_obj']['x1']
                            y1 = wire_item['wire_obj']['y1']
                            x2 = wire_item['wire_obj']['x2']
                            y2 = wire_item['wire_obj']['y2']
```

```
                            # Test to see if wire obj matches wire coordinates
                            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                                conn = Connection(switch, wire,
wire_item['wire_end'])
                                switch.wire_list.append(conn)

                    elif item['type'] == "led":
                        wire_list = item['wire_list']
                        for wire_item in wire_list:
                            x1 = wire_item['wire_obj']['x1']
                            y1 = wire_item['wire_obj']['y1']
                            x2 = wire_item['wire_obj']['x2']
                            y2 = wire_item['wire_obj']['y2']
                            # Test to see if wire obj matches wire coordinates
                            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                                conn = Connection(led, wire,
wire_item['wire_end'])
                                led.wire_list.append(conn)

        except FileNotFoundError:
            with open('untitled.canvas', 'w') as _file:
                pass
            self.canvas.comp_list = []

    def save_diagram(self):  # Modified to save the canvas comp list as a json
file
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            file.write(json.dumps(self.canvas.comp_list, cls=MyEncoder,
indent=4))

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False
```

diagram1.json

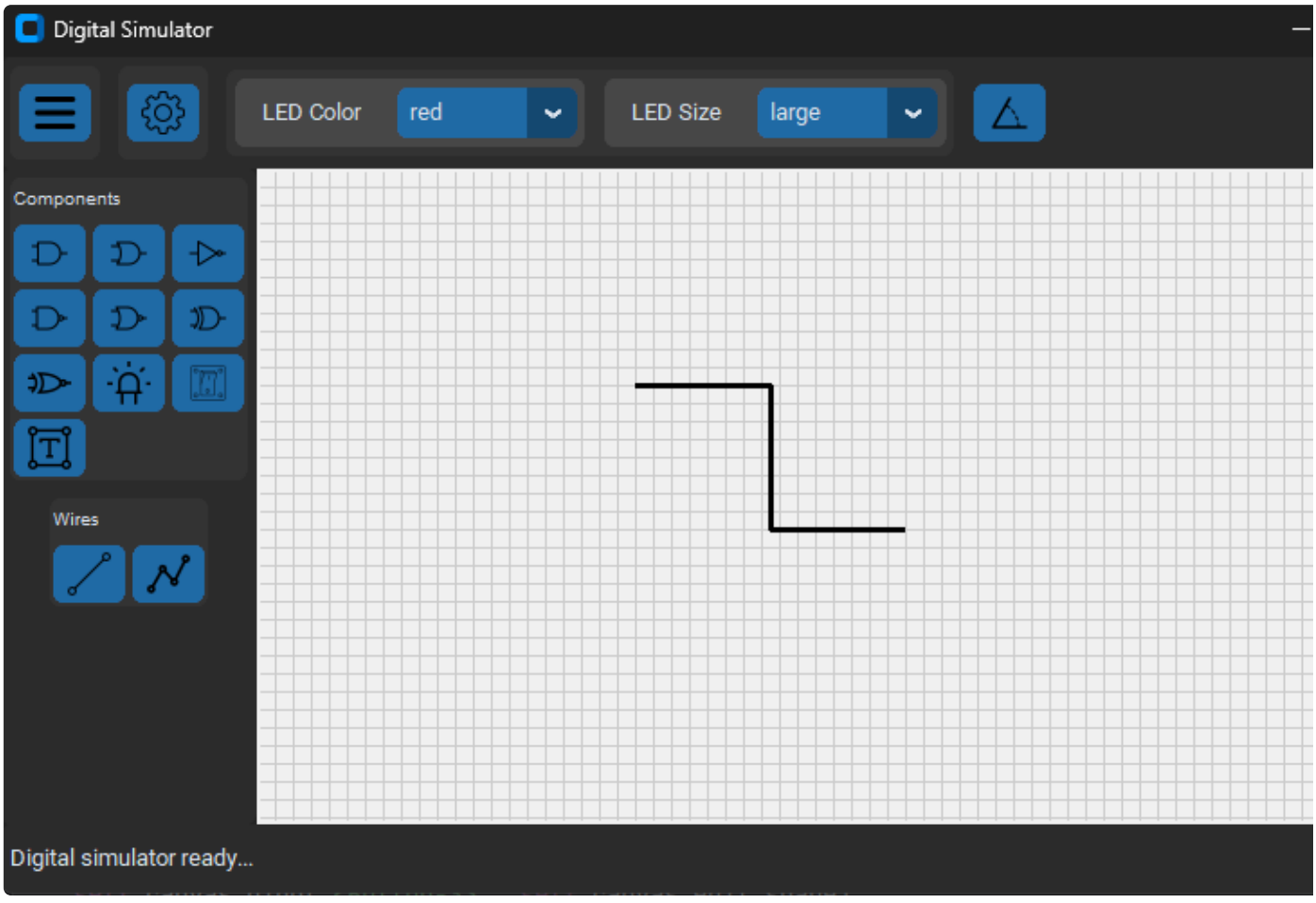
```
[
    {
        "type": "switch",
        "x1": 160,
        "y1": 200,
        "angle": 0,
        "wire_list": [
            {
                "type": "connection",
                "wire_obj": {
                    "type": "wire",
                    "x1": 180.0,
                    "y1": 200.0,
                    "x2": 358.0,
                    "y2": 200.0
                },
                "wire_end": "begin"
            }
        ]
    },
    {
        "type": "led",
        "x1": 380,
        "y1": 200,
        "angle": 180,
        "wire_list": [
            {
                "type": "connection",
                "wire_obj": {
                    "type": "wire",
                    "x1": 180.0,
                    "y1": 200.0,
                    "x2": 358.0,
                    "y2": 200.0
                },
                "wire_end": "end"
            }
        ]
    },
    {
        "type": "wire",
        "x1": 180.0,
        "y1": 200.0,
        "x2": 358.0,
        "y2": 200.0
```

```
    }
]
```

Segment Wire Class

Objectives:

- Create a 3-segment wire class
- Configure the segment wire for horizontal or vertical orientation



Wire_Lib/segment_wire.py - new class

```
from Wire_Lib.wire_selector import WireSelector


class SegmentWire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """3-Segment Wire"""
```

```
        self.type = "segment_wire"
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = self.canvas.line_direction

        self.id = None
        self.state = False

        self.bbox = None
        self.is_selected = False
        self.sel_list = []
        self.segment_list = None
        self.points = None
        self.selector = None

        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.create_segmented_line()

        self.s1_id, self.s2_id = None, None
        self.create_selectors()
        self.set_selector_visibility()

    def create_segmented_line(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        segment1, segment2, segment3 = None, None, None

        if self.line_direction == "horizontal":
            segment1 = self.x1, self.y1, self.x1 + w/2, self.y1
            segment2 = self.x1 + w/2, self.y1, self.x1 + w/2, self.y2
            segment3 = self.x1 + w/2, self.y2, self.x2, self.y2
        elif self.line_direction == "vertical":
            segment1 = self.x1, self.y1, self.x1, self.y1 + h/2
            segment2 = self.x1, self.y1 + h/2, self.x2, self.y1 + h/2
            segment3 = self.x2, self.y1 + h/2, self.x2, self.y2
        self.segment_list = [segment1, segment2, segment3]
        self.draw_segments()

    def draw_segments(self):
```

```
        for s in self.segment_list:
            if self.state:
                self.id = self.canvas.create_line(s, fill="#00ff00",
width=self.border_width, tags='wire')
            else:
                self.id = self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width, tags='wire')

    def create_selectors(self):
        """Create selectors at the ends of the wire here"""
        self.s1_id = WireSelector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = WireSelector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_wire_color()
        self.update_selectors()
        self.set_selector_visibility()

    def update_position(self):
        """Update the position when the gate object is moved"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        if self.line_direction == "horizontal":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2,
self.y1,
                               self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
                               self.x1 + w / 2, self.y2, self.x2, self.y2)
        elif self.line_direction == "vertical":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                               self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                               self.x2, self.y1 + h / 2, self.x2, self.y2)

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_wire_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="#00ff00")
        else:
```

```
            self.canvas.itemconfig(self.id, fill=self.fill_color)

    def update_border_width(self):
        self.canvas.itemconfig(self.id, width=self.border_width)

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def set_selector_visibility(self):
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
            # self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2,
self.y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
            # self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1,
self.y1)

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Wire: " + " x1: " + str(self.x1) + " y1: " + str(self.y1) + \
            " x2: " + str(self.x2) + " y2: " + str(self.y2)
```

```
    def reprJson(self):
        return dict(type=self.type, x1=self.x1, y1=self.y1, x2=self.x2,
y2=self.y2)
```

UI_Lib/canvas.py

```
. . .

class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.led_color = "red"
        self.led_size = "large"
        self.line_direction = "horizontal"  # Added variable to store line
direction for segment wires

. . .
```

UI_Lib/wire_button_frame.py

```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import Wire, SegmentWire  # Added import for segment wire class


class WireButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.wire = None

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Wires", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)
```

```
        straight_wire_image = ctk.CTkImage(light_image=Image.open  # change
variable name to straight wire
                                   (Path(__file__).parent /
"../icons/straight_line.png"),
                                   dark_image=Image.open
                                   (Path(__file__).parent /
"../icons/straight_line.png"),
                                   size=(24, 24))

        straight_wire_button = ctk.CTkButton(self, text="",
image=straight_wire_image, width=30,    # change variable name to straight wire
                                           command=self.create_straight_wire)
        straight_wire_button.grid(row=1, column=0, sticky=ctk.W, padx=2,
pady=2)    # change variable name to straight wire

        segment_wire_image = ctk.CTkImage(light_image=Image.open  # Add segment
wire imate
                                   (Path(__file__).parent /
"../icons/segment_line.png"),
                                   dark_image=Image.open
                                   (Path(__file__).parent /
"../icons/segment_line.png"),
                                   size=(24, 24))

        segment_wire_button = ctk.CTkButton(self, text="",
image=segment_wire_image, width=30,  # Add segment wire button
                                           command=self.create_segment_wire)
        segment_wire_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
# Add segment wire button

    # Shape button handlers
    def create_straight_wire(self):
        wire = Wire(self.canvas, 0, 0, 0, 0)
        self.canvas.comp_list.append(wire)
        self.canvas.mouse.current_wire_obj = wire
        self.canvas.show_connectors()
        self.canvas.mouse.draw_wire_mouse_events()

    def create_segment_wire(self):  # Added method to create segment wire
        print("Create segment wire called")
        wire = SegmentWire(self.canvas, 0, 0, 0, 0)
        self.canvas.comp_list.append(wire)
        self.canvas.mouse.current_wire_obj = wire
        self.canvas.show_connectors()
```

```
        self.canvas.mouse.draw_wire_mouse_events()
```

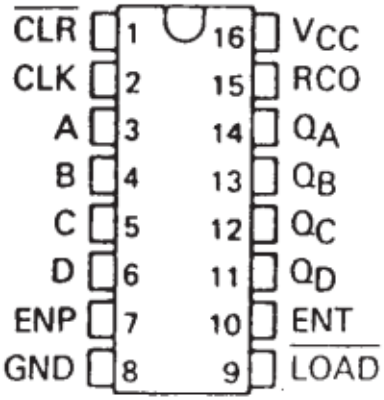
74ls161 Counter IC Class

Objectives:

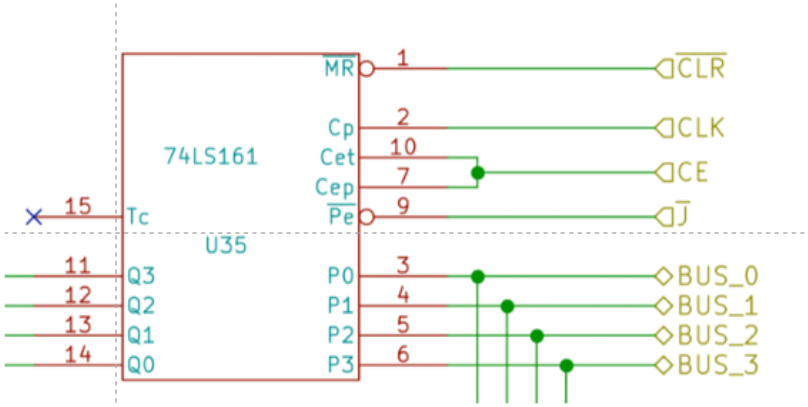
- Get the pin-out and logic diagram for 74ls161 IC
- Draw the image of the IC
- Create a 74161 IC class
- Add an IC frame to the Left Frame Class
- Create the logic for the 74161 counter

74ls161 Data Sheet

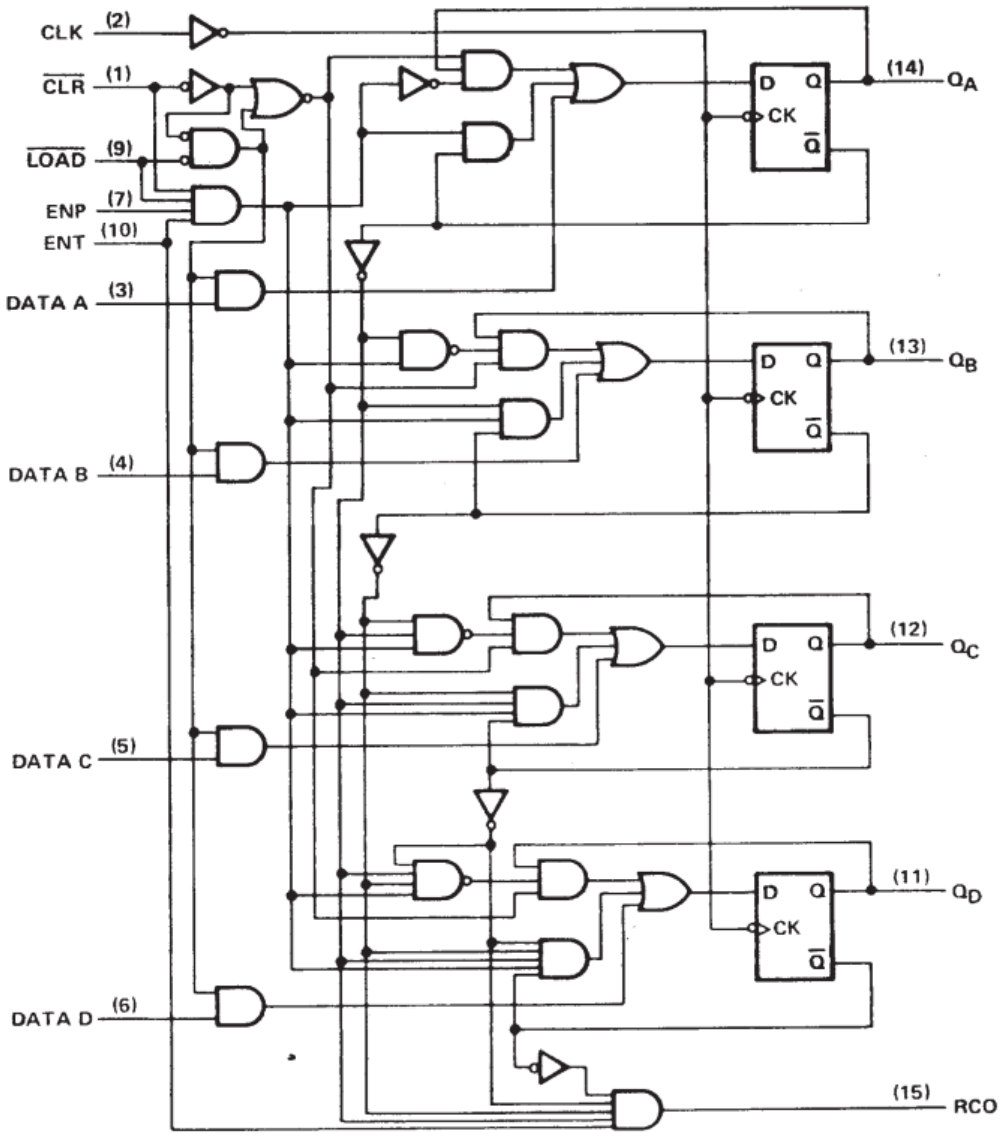
Device pin-out



Device alternate pin-out for simulation

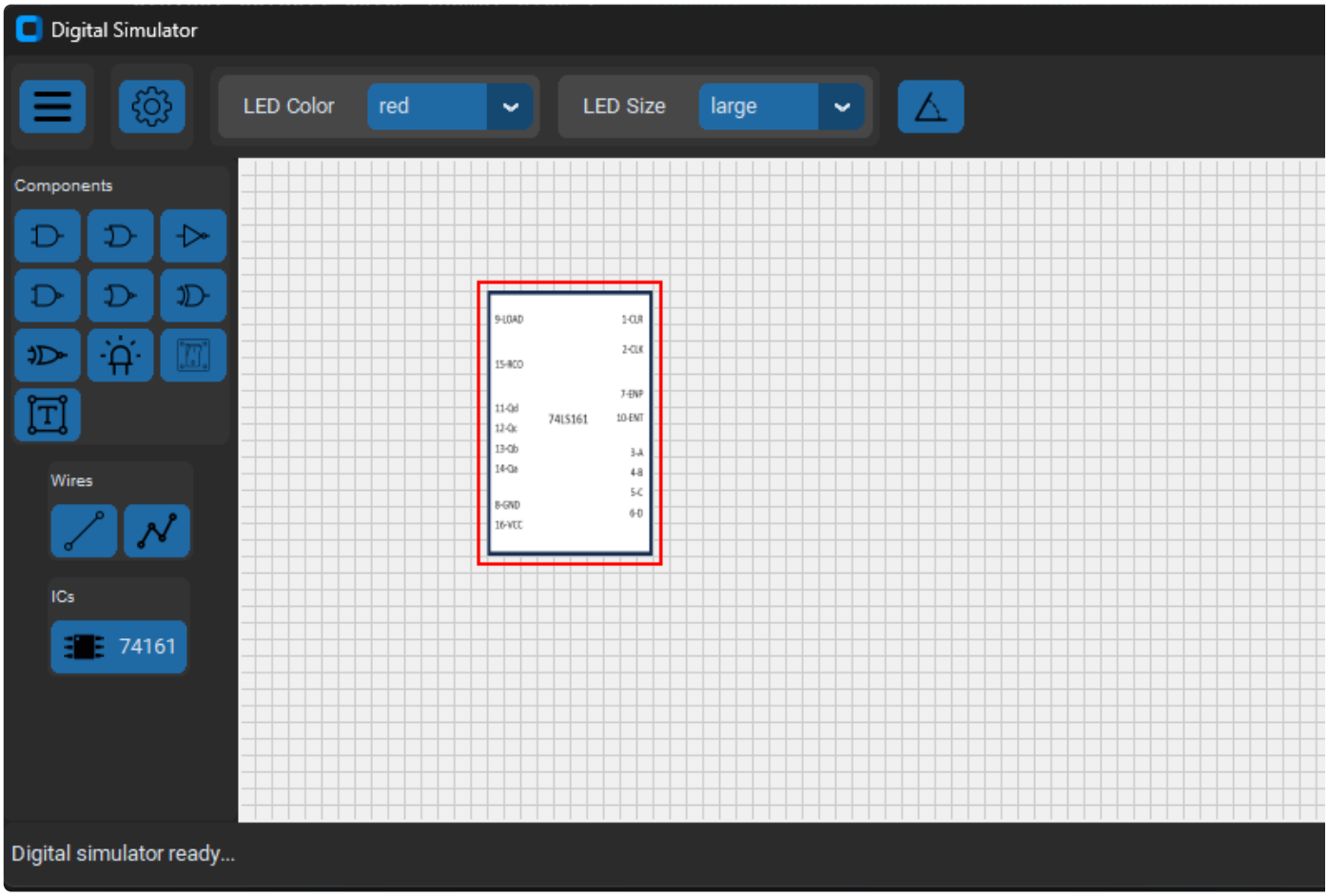


Device logic diagram



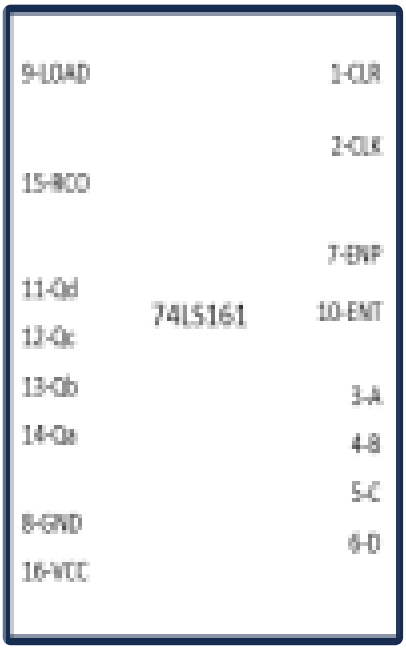
Notes:

- Data A, B, C, D (pins 3, 4, 5, 6) - input data to set counter to known value if |LOAD (pin 9) is enabled (set to False)
- `Qa, Qb, Qc, Qd` (pins 14, 13, 12, 11) - output the current counter data
- ENP, ENT (pins 7, 10) - Chip enable
- CLK (pin 2) - Clock input
- |CLR (pin 1) - Clear input (enabled when set to False)
- RCO (pin 15) - Not used
- GND (Pin 8) - power ground
- VCC (Pin 16) - power voltage



74LS161 Counter Image

- Drawn in PowerPoint
- Copied to Paint3D
- Resized to w=100 and h=160



IC_Lib/IC.py - IC base class - new class

```
import tkinter as tk
from PIL import Image, ImageTk


class IC:
    def __init__(self, canvas, x1, y1):
        """Base class for integrated circuit (IC) classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
        self.filename = None

        self.is_selected = False
        self.is_drawing = False

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.wire_list = []
```

```
    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

    def update_position(self):
        """Update the position when the gate object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1)  # Update position

    def update_image(self, filename):
        """Update the image for gate symbol rotation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)  # Update
image rotation
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.canvas.itemconfig(self.id, image=self.ph_image)  # Update image

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
        self.set_selector_visibility()

    def update_selector(self):
        """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.canvas.coords(self.sel_id, x1, y1, x2, y2)
        self.set_selector_visibility()

    def set_selector_visibility(self):
        """Set the selector visibility state"""
        if self.is_selected:
            self.canvas.itemconfig(self.sel_id, state='normal')
        else:
```

```
            self.canvas.itemconfig(self.sel_id, state='hidden')

    def set_connector_visibility(self):
        """Set the connector visibility state"""
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

    def rotate(self):
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def check_connector_hit(self, x, y):
        """Hit test to see if a connector is at the provided x, y
coordinates"""
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_wires(self):
        """Resize connected wires if the shape is moved"""
        for connection in self.wire_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.wire_end == "begin":
                        connection.wire_obj.x1 = connector.x
                        connection.wire_obj.y1 = connector.y
                    elif connection.wire_end == "end":
                        connection.wire_obj.x2 = connector.x
                        connection.wire_obj.y2 = connector.y
```

IC_Lib/ic74161_counter.py

```
from pathlib import Path
```

```
from IC_Lib.ic import IC


class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.filename = Path(__file__).parent /
"../images/ics/74161_easy_100x160.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        # self.create_connectors()
        # self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        # self.update_connectors()
        # self.set_connector_visibility()
```

UI_Lib/ic_button_frame.py - new class

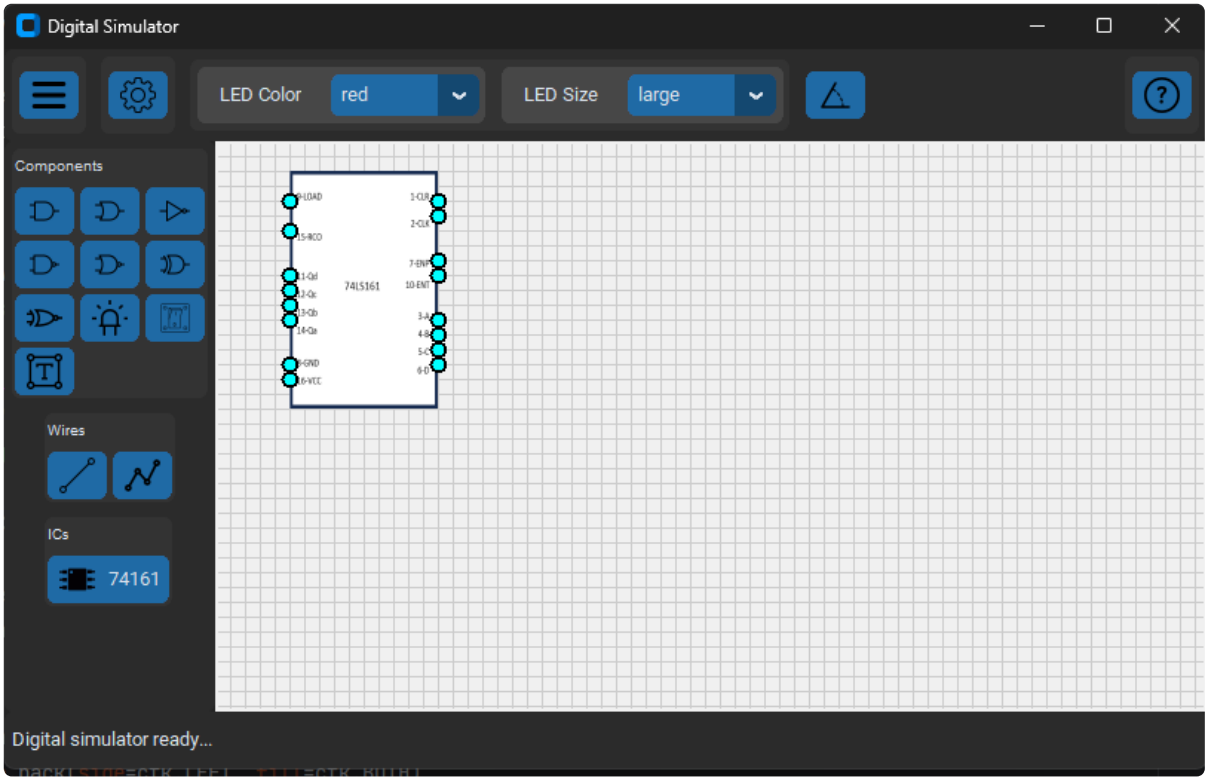
```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from IC_Lib import IC74161


class ICButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="ICs", font=("Helvetica",
10), height=20)
```

```
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        ic_image = ctk.CTkImage(light_image=Image.open
                (Path(__file__).parent / "../icons/ic.png"),
               dark_image=Image.open
               (Path(__file__).parent / "../icons/ic.png"),
               size=(24, 24))

        ic_button = ctk.CTkButton(self, text="74161", image=ic_image, width=30,
                                            command=self.create_ic_74161)
        ic_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

    def create_ic_74161(self):
        ic = IC74161(self.canvas, 100, 100)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()
```

UI_Lib/left_frame.py

```
import customtkinter as ctk
from UI_Lib.comp_button_frame import CompButtonFrame
from UI_Lib.wire_button_frame import WireButtonFrame
from UI_Lib.ic_button_frame import ICButtonFrame


class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.comp_button_frame = CompButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.wire_button_frame = WireButtonFrame(self, self.canvas)
        self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.ic_button_frame = ICButtonFrame(self, self.canvas)
        self.ic_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

74LS161 IC Connectors



IC_Lib/ic74161_counter.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}

        # Set initial logic states
        for i in range(1, 17):  # 16 pin IC
            self.logic_dict['c' + str(i)] = False
        self.logic_dict['c14'] = True

        self.filename = Path(__file__).parent /
```

```
"../images/ics/74161_easy_100x160.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()  # Added call to create connectors
        self.set_connector_visibility()  # Added call to set connector
visibility

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()  # Added call to update connectors
        self.set_connector_visibility()  # Added call to set connector
visibility

    def create_connectors(self):  # Added new method to create connectors
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.conn_list.append(Connector(self.canvas, "c1", center.x + w/2,
center.y - h/2 + 20))
        self.conn_list.append(Connector(self.canvas, "c2", center.x + w / 2,
center.y - h / 2 + 30))

        self.conn_list.append(Connector(self.canvas, "c3", center.x + w / 2,
center.y - h / 2 + 100))
        self.conn_list.append(Connector(self.canvas, "c4", center.x + w / 2,
center.y - h / 2 + 110))
        self.conn_list.append(Connector(self.canvas, "c5", center.x + w / 2,
center.y - h / 2 + 120))
        self.conn_list.append(Connector(self.canvas, "c6", center.x + w / 2,
center.y - h / 2 + 130))

        self.conn_list.append(Connector(self.canvas, "c7", center.x + w / 2,
center.y - h / 2 + 60))
        self.conn_list.append(Connector(self.canvas, "c10", center.x + w / 2,
center.y - h / 2 + 70))

        self.conn_list.append(Connector(self.canvas, "c9", center.x - w / 2,
center.y - h/2 + 20))
        self.conn_list.append(Connector(self.canvas, "c15", center.x - w / 2,
center.y - h / 2 + 40))
```

```
        self.conn_list.append(Connector(self.canvas, "c11", center.x - w / 2,
center.y - h / 2 + 70))
        self.conn_list.append(Connector(self.canvas, "c12", center.x - w / 2,
center.y - h / 2 + 80))
        self.conn_list.append(Connector(self.canvas, "c13", center.x - w / 2,
center.y - h / 2 + 90))
        self.conn_list.append(Connector(self.canvas, "c14", center.x - w / 2,
center.y - h / 2 + 100))
        self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 + 130))
        self.conn_list.append(Connector(self.canvas, "c16", center.x - w / 2,
center.y - h / 2 + 140))

    def update_connectors(self):  # Added new method to update connectors
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.conn_list[0].x, self.conn_list[0].y = center.x + w/2, center.y -
h/2 + 20
        self.conn_list[1].x, self.conn_list[1].y = center.x + w / 2, center.y -
h / 2 + 30
        self.conn_list[2].x, self.conn_list[2].y = center.x + w / 2, center.y -
h / 2 + 100
        self.conn_list[3].x, self.conn_list[3].y = center.x + w / 2, center.y -
h / 2 + 110
        self.conn_list[4].x, self.conn_list[4].y = center.x + w / 2, center.y -
h / 2 + 120
        self.conn_list[5].x, self.conn_list[5].y = center.x + w / 2, center.y -
h / 2 + 130
        self.conn_list[6].x, self.conn_list[6].y = center.x + w / 2, center.y -
h / 2 + 60
        self.conn_list[7].x, self.conn_list[7].y = center.x + w / 2, center.y -
h / 2 + 70

        self.conn_list[8].x, self.conn_list[8].y = center.x - w/2, center.y -
h/2 + 20
        self.conn_list[9].x, self.conn_list[9].y = center.x - w / 2, center.y -
h / 2 + 40
        self.conn_list[10].x, self.conn_list[10].y = center.x - w / 2, center.y
- h / 2 + 70
        self.conn_list[11].x, self.conn_list[11].y = center.x - w / 2, center.y
- h / 2 + 80
        self.conn_list[12].x, self.conn_list[12].y = center.x - w / 2, center.y
```

```
- h / 2 + 90
        self.conn_list[13].x, self.conn_list[13].y = center.x - w / 2, center.y
- h / 2 + 100
        self.conn_list[14].x, self.conn_list[14].y = center.x - w / 2, center.y
- h / 2 + 130
        self.conn_list[15].x, self.conn_list[15].y = center.x - w / 2, center.y
- h / 2 + 140

        # Draw the connectors
        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        # c1 = |CLR
        # c2 = CLK
        # c3 = A
        # c4 = B
        # c5 = C
        # c6 = D
        # c7 = ENP
        # c8 = GND
        # c9 = |LOAD
        # c10 = ENT
        # c11 = Qd
        # c12 = Qc
        # c13 = Qd
        # c14 = Qa
        # C15 = RCO
        # C16 = VCC

        for wire in self.wire_list:
            # NAND Gate #1
            if wire.connector_obj.name == "c1":
                self.logic_dict['c1'] = wire.line_obj.state
                self.logic_dict['c3'] = not(self.logic_dict['c1'] and
self.logic_dict['c2'])
            elif wire.connector_obj.name == "c2":
                self.logic_dict['c2'] = wire.line_obj.state
                self.logic_dict['c3'] = not(self.logic_dict['c1'] and
self.logic_dict['c2'])
            elif wire.connector_obj.name == "c3":
                wire.line_obj.state = self.logic_dict['c3']

            # NAND Gate #2
```

```
            elif wire.connector_obj.name == "c4":
                self.logic_dict['c4'] = wire.line_obj.state
                self.logic_dict['c6'] = not(self.logic_dict['c4'] and
self.logic_dict['c5'])
            elif wire.connector_obj.name == "c5":
                self.logic_dict['c5'] = wire.line_obj.state
                self.logic_dict['c6'] = not(self.logic_dict['c4'] and
self.logic_dict['c5'])
            elif wire.connector_obj.name == "c6":
                wire.line_obj.state = self.logic_dict['c6']

            # NAND Gate #3
            elif wire.connector_obj.name == "c10":
                self.logic_dict['c10'] = wire.line_obj.state
                self.logic_dict['c8'] = not(self.logic_dict['c10'] and
self.logic_dict['c9'])
            elif wire.connector_obj.name == "c9":
                self.logic_dict['c9'] = wire.line_obj.state
                self.logic_dict['c8'] = not(self.logic_dict['c10'] and
self.logic_dict['c9'])
            elif wire.connector_obj.name == "c8":
                wire.line_obj.state = self.logic_dict['c8']

            # NAND Gate #4
            elif wire.connector_obj.name == "c13":
                self.logic_dict['c13'] = wire.line_obj.state
                self.logic_dict['c11'] = not(self.logic_dict['c13'] and
self.logic_dict['c12'])
            elif wire.connector_obj.name == "c12":
                self.logic_dict['c12'] = wire.line_obj.state
                self.logic_dict['c11'] = not(self.logic_dict['c13'] and
self.logic_dict['c12'])
            elif wire.connector_obj.name == "c11":
                wire.line_obj.state = self.logic_dict['c11']
```

74LS161 Logic Levels

Start by testing a binary counter
Sandbox/test_binary_count.py

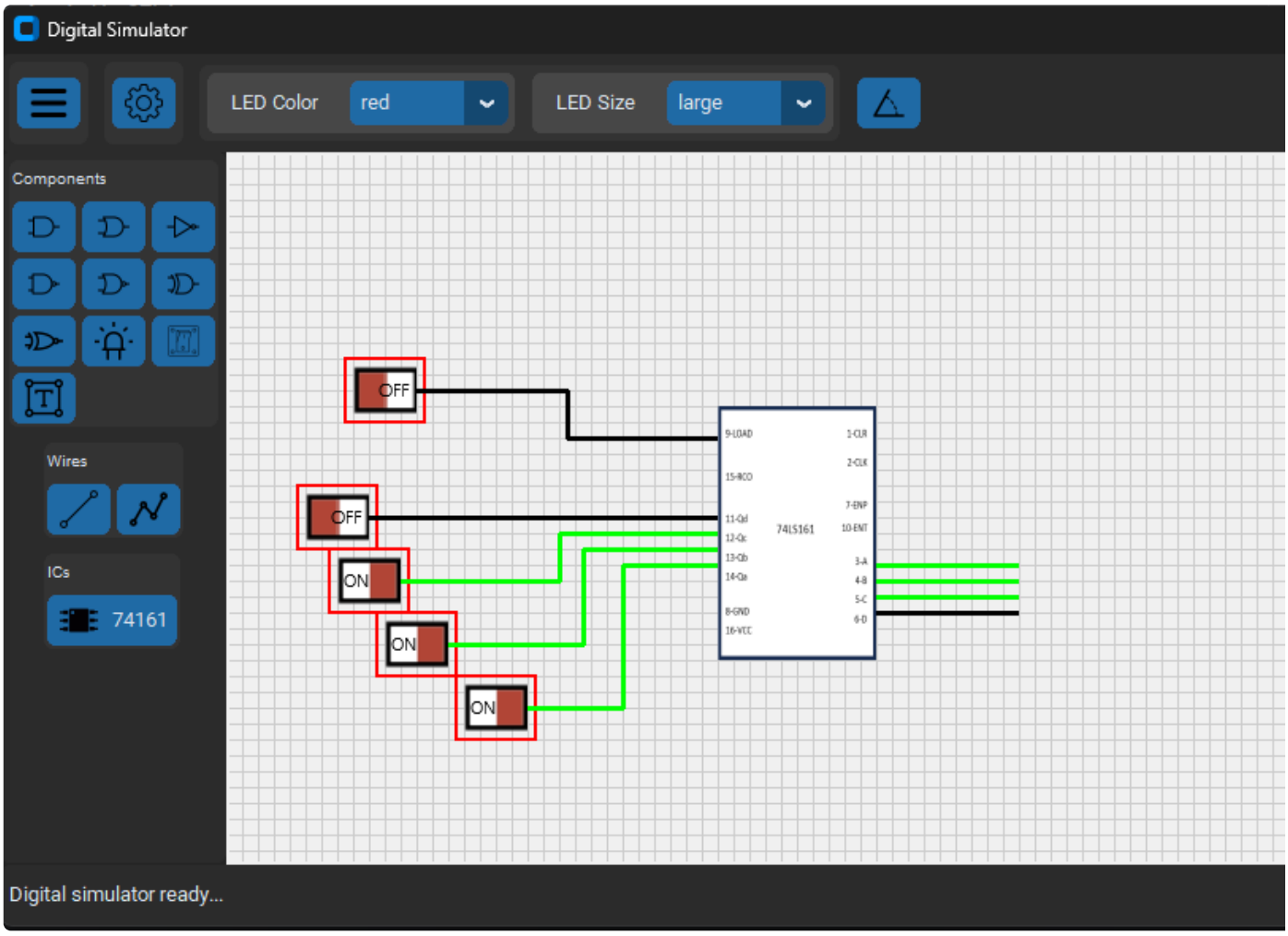
```
def add_one(num):
    num += 1
    if num > 15:
        num = 0
    return num


def parse_binary(binary):
    d1 = binary[3]
    d2 = binary[2]
    d3 = binary[1]
    d4 = binary[0]
    print("d1: ", d1, " d2: ", d2, " d3: ", d3, " d4: ", d4)


num_int = 0
for i in range(1, 16):
    num_int = add_one(num_int)
    binary = format(num_int, '04b')
    print(binary)
    parse_binary(binary)
```

Console Output:

```
0001
d1:  1  d2:  0  d3:  0  d4:  0
0010
d1:  0  d2:  1  d3:  0  d4:  0
0011
d1:  1  d2:  1  d3:  0  d4:  0
0100
d1:  0  d2:  0  d3:  1  d4:  0
0101
d1:  1  d2:  0  d3:  1  d4:  0
0110
d1:  0  d2:  1  d3:  1  d4:  0
0111
d1:  1  d2:  1  d3:  1  d4:  0
1000
d1:  0  d2:  0  d3:  0  d4:  1
```

```
1001
d1:  1  d2:  0  d3:  0  d4:  1
1010
d1:  0  d2:  1  d3:  0  d4:  1
1011
d1:  1  d2:  1  d3:  0  d4:  1
1100
d1:  0  d2:  0  d3:  1  d4:  1
1101
d1:  1  d2:  0  d3:  1  d4:  1
1110
d1:  0  d2:  1  d3:  1  d4:  1
1111
d1:  1  d2:  1  d3:  1  d4:  1
```

Load Test



Test: Good

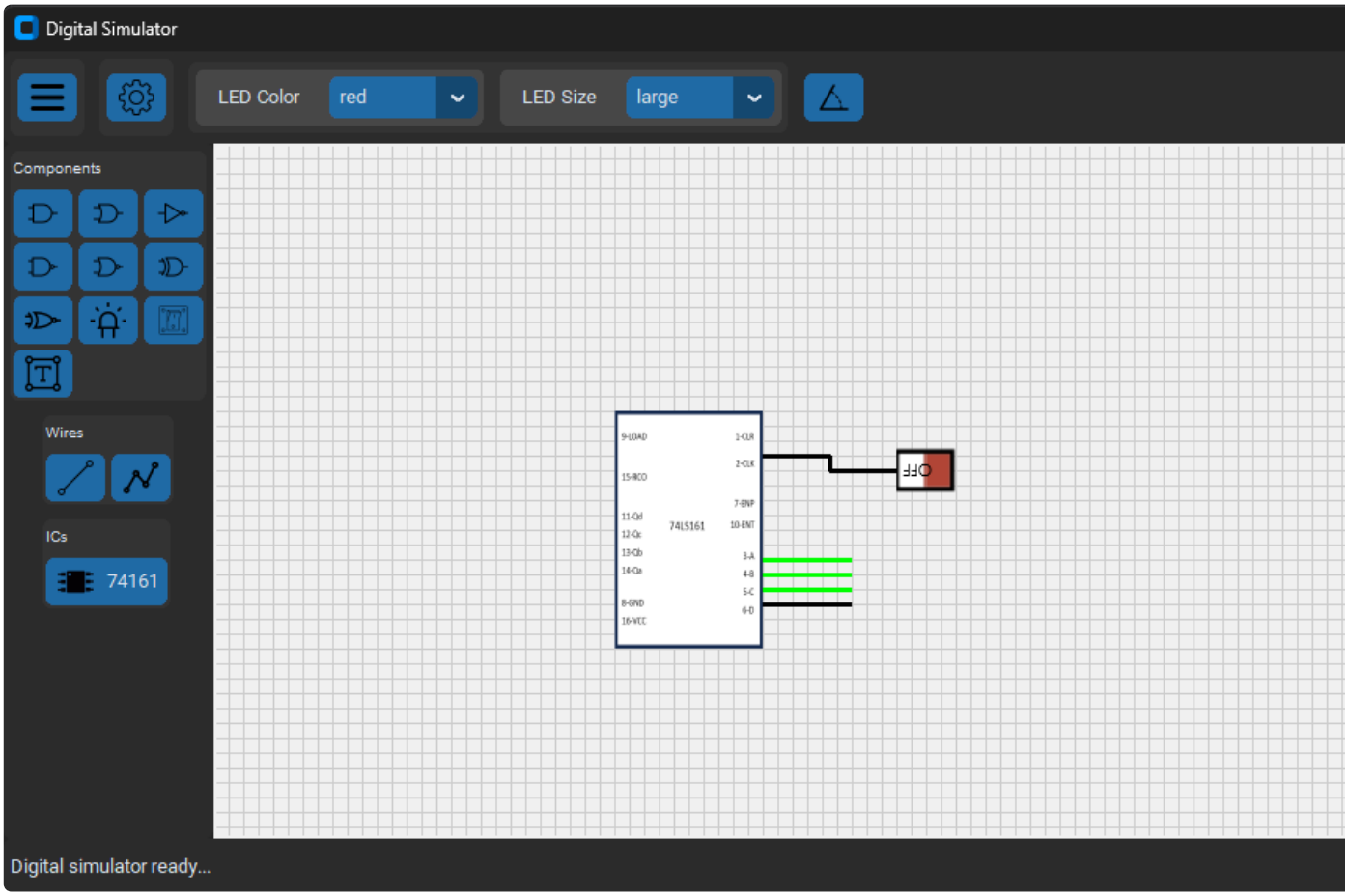
IC_Lib/ic74161_counter.py

```
. . .

class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}
        self.count_int = 0  # Added a variable to store counter value as
integer

        # Initialize 4 x D-flip flops
        self.d1, self.d2, self.d3, self.d4 = False, False, False, False  #
Initialize the 4 x D-flip flops

        # Set initial logic states
        for i in range(1, 17):  # 16 pin IC
            self.logic_dict['c' + str(i)] = False

. . .

    def update(self):
        self.set_logic_level()  # Added call to set logic level
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

. . .

    def set_logic_level(self):  # Added new method to set logic levels
        # c1 = |CLR
        # c2 = CLK
        # c3 = A
        # c4 = B
        # c5 = C
        # c6 = D
        # c7 = ENP
        # c8 = GND
        # c9 = |LOAD
```

```
        # c10 = ENT
        # c11 = Qd
        # c12 = Qc
        # c13 = Qd
        # c14 = Qa
        # C15 = RCO
        # C16 = VCC

        for wire in self.wire_list:
            # NAND Gate #1
            if wire.connector_obj.name == "c1":  # CLR input
                self.logic_dict['c1'] = wire.wire_obj.state
                if not wire.wire_obj.state:
                    self.d1 = self.d2 = self.d3 = self.d4 = False
                    self.set_ABCD()
            elif wire.connector_obj.name == "c2":  # CLK input
                self.logic_dict['c2'] = wire.wire_obj.state
                if wire.wire_obj.state:
                    self.count_int = self.add_one(self.count_int)
                    binary = format(self.count_int, '04b')
                    self.parse_binary(binary)
                    self.set_ABCD()
            elif wire.connector_obj.name == "c3":  # Output A
                wire.wire_obj.state = self.logic_dict['c3']
            elif wire.connector_obj.name == "c4":  # Output B
                wire.wire_obj.state = self.logic_dict['c4']
            elif wire.connector_obj.name == "c5":  # Output C
                wire.wire_obj.state = self.logic_dict['c5']
            elif wire.connector_obj.name == "c6":  # Output D
                wire.wire_obj.state = self.logic_dict['c6']
            elif wire.connector_obj.name == "c7":  # ENP - enable input
                self.logic_dict['c7'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c10":  # ENT - enable input
                self.logic_dict['c10'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c8":  # GND - input
                self.logic_dict['c8'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c16":  # VCC - input
                self.logic_dict['c16'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c9":  # |LOAD - input
                self.logic_dict['c9'] = wire.wire_obj.state
                if not wire.wire_obj.state:
                    self.d1 = self.logic_dict["c14"]
                    self.d2 = self.logic_dict["c13"]
                    self.d3 = self.logic_dict["c12"]
                    self.d4 = self.logic_dict["c11"]
                    self.set_ABCD()
```

```
            elif wire.connector_obj.name == "c11":  # Qd - input
                self.logic_dict['c11'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c12":  # Qc - input
                self.logic_dict['c12'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c13":  # Qb - input
                self.logic_dict['c13'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c14":  # Qa - input
                self.logic_dict['c14'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c15":  # RCO - input
                self.logic_dict['c15'] = wire.wire_obj.state

    def set_ABCD(self):  # Helper method to set the IC outputs
        self.logic_dict['c3'] = self.d1
        self.logic_dict['c4'] = self.d2
        self.logic_dict['c5'] = self.d3
        self.logic_dict['c6'] = self.d4

    @staticmethod
    def add_one(num):  # Helper method to add numbers up to 15 then reset to 0
        num += 1
        if num > 15:
            num = 0
        return num

    def parse_binary(self, bin_num):  # Helper method to set D flip-flops to a
4-bit binary number
        self.d1 = bin_num[3]
        self.d2 = bin_num[2]
        self.d3 = bin_num[1]
        self.d4 = bin_num[0]
```

Clock Test



Watch out for binary 1/0 to True/False mismatches.

IC_Lib/ic74161_counter.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}
        self.count_int = 0  # Added variable to store count

        # Initialize 4 x D-flip flops
```

```
        self.d1, self.d2, self.d3, self.d4 = False, False, False, False  # Initialize the 4xD flip-flops

        # Set initial logic states
        for i in range(1, 17):  # 16 pin IC
            self.logic_dict['c' + str(i)] = False

        self.filename = Path(__file__).parent / "../images/ics/74161_easy_100x160.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    . . .

    def set_logic_level(self):
        # c1 = |CLR
        # c2 = CLK
        # c3 = A
        # c4 = B
        # c5 = C
        # c6 = D
        # c7 = ENP
        # c8 = GND
        # c9 = |LOAD
        # c10 = ENT
        # c11 = Qd
        # c12 = Qc
        # c13 = Qd
        # c14 = Qa
        # C15 = RCO
        # C16 = VCC

        for wire in self.wire_list:
            # print(wire)
            if wire.connector_obj.name == "c1":  # CLR input
                self.logic_dict['c1'] = wire.wire_obj.state
                if not wire.wire_obj.state:
                    self.d1 = self.d2 = self.d3 = self.d4 = False
                    self.set_ABCD()
            elif wire.connector_obj.name == "c2":  # CLK input
                self.logic_dict['c2'] = wire.wire_obj.state
                if wire.wire_obj.state:
                    self.count_int = self.add_one(self.count_int)
```

```
                    binary = format(self.count_int, '04b')
                    self.parse_binary(binary)
                    self.set_ABCD()
            elif wire.connector_obj.name == "c3":  # Output A
                wire.wire_obj.state = self.logic_dict['c3']
                # print("c3 wire: ", wire.wire_obj, " state ",
wire.wire_obj.state)
            elif wire.connector_obj.name == "c4":  # Output B
                wire.wire_obj.state = self.logic_dict['c4']
                # print("c4 wire: ", wire.wire_obj, " state ",
wire.wire_obj.state)
            elif wire.connector_obj.name == "c5":  # Output C
                wire.wire_obj.state = self.logic_dict['c5']
                # print("c5 wire: ", wire.wire_obj, " state ",
wire.wire_obj.state)
            elif wire.connector_obj.name == "c6":  # Output D
                wire.wire_obj.state = self.logic_dict['c6']
                # print("c6 wire: ", wire.wire_obj, " state ",
wire.wire_obj.state)
            elif wire.connector_obj.name == "c7":  # ENP - enable input
                self.logic_dict['c7'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c10":  # ENT - enable input
                self.logic_dict['c10'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c8":  # GND - input
                self.logic_dict['c8'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c16":  # VCC - input
                self.logic_dict['c16'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c9":  # |LOAD - input
                self.logic_dict['c9'] = wire.wire_obj.state
                if not wire.wire_obj.state:
                    self.d1 = self.logic_dict["c14"]
                    self.d2 = self.logic_dict["c13"]
                    self.d3 = self.logic_dict["c12"]
                    self.d4 = self.logic_dict["c11"]
                    self.set_ABCD()
            elif wire.connector_obj.name == "c11":  # Qd - input
                self.logic_dict['c11'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c12":  # Qc - input
                self.logic_dict['c12'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c13":  # Qb - input
                self.logic_dict['c13'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c14":  # Qa - input
                self.logic_dict['c14'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c15":  # RCO - input
                self.logic_dict['c15'] = wire.wire_obj.state
```

```
     def set_ABCD(self):
        self.logic_dict['c3'] = self.d1
        self.logic_dict['c4'] = self.d2
        self.logic_dict['c5'] = self.d3
        self.logic_dict['c6'] = self.d4

    @staticmethod
    def add_one(num):
        num += 1
        if num > 15:
            num = 0
        return num

    def parse_binary(self, bin_num):
def parse_binary(self, bin_num):
    result = lambda s: True if s == '1' else False
    self.d1 = result(bin_num[3])
    self.d2 = result(bin_num[2])
    self.d3 = result(bin_num[1])
    self.d4 = result(bin_num[0])
```

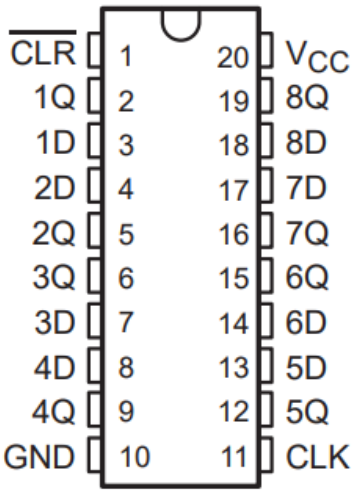
74ls273 Octal D Flip-Flops Class

Objectives:

- Create a new 74273 IC class
- Create a new 74273 image file with "Ds" on one side and "Qs" on the other side
- Test the IC as a storage register

74LS273 Data Sheet

Device pin-out

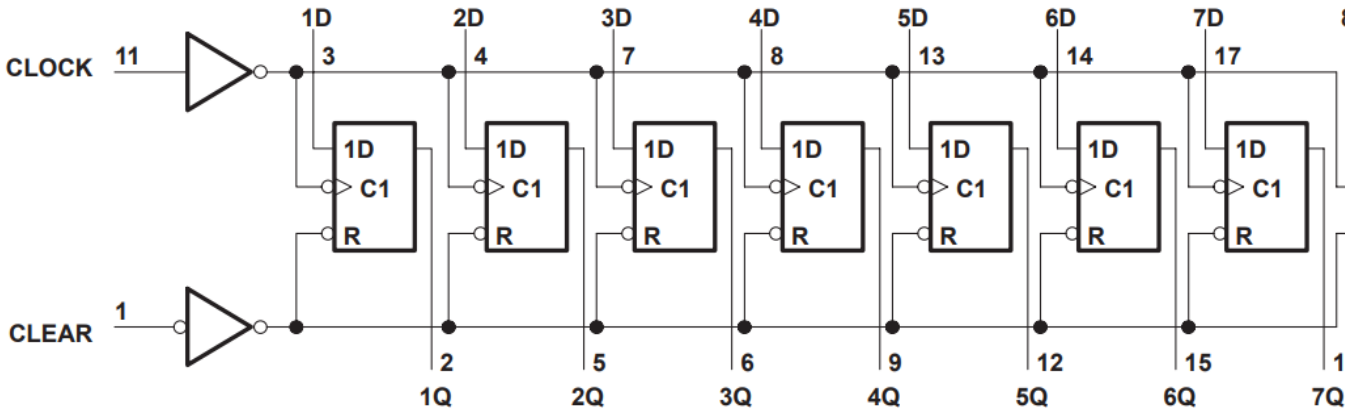


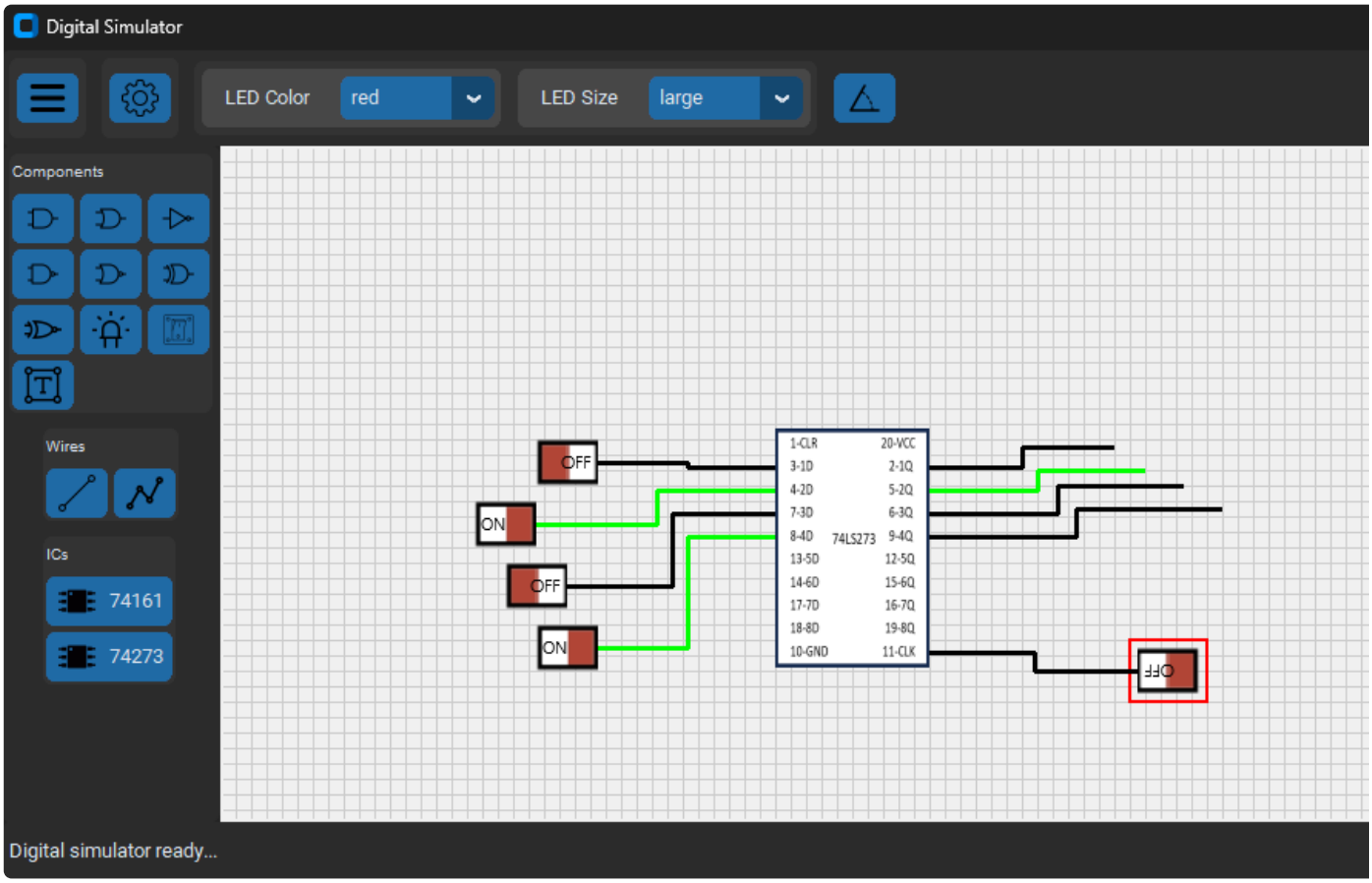
Function Table

FUNCTION TABLE
(each flip-flop)

INPUTS			OUTPUT
CLEAR	CLOCK	D	Q
L	X	X	L
H	↑	H	H
H	↑	L	L
H	L	X	Q_0

Logic Diagram





IC_Lib/ic_74273_flip_flop.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class DFlipFlop:
    """Logical model for D Flip-Flop"""
    def __init__(self):
        self.R = False  # Clear
        self.C1 = False  # Clock
        self.D1 = False  # D Input
        self.Q1 = False  # Q Output

    def clear_ic(self):
        self.Q1 = False

    def clock_high(self):
        self.Q1 = self.D1
```

```
    def __repr__(self):
        return ("D Flip-Flop: " + "R: " + str(self.R) + " C1: " + str(self.C1)
+
                " D1: " + str(self.D1) + " Q1: " + str(self.Q1))


class IC74273(IC):
    """Model for 74ls273 Quad D Flip-Flop IC - 20-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74273"

        self.logic_dict = {}
        self.count_int = 0
        self.conn_inc = 15
        self.offset = -5

        # Initialize 8 x D-flip flops
        self.d1 = DFlipFlop()
        self.d2 = DFlipFlop()
        self.d3 = DFlipFlop()
        self.d4 = DFlipFlop()
        self.d5 = DFlipFlop()
        self.d6 = DFlipFlop()
        self.d7 = DFlipFlop()
        self.d8 = DFlipFlop()
        self.ff_list = [self.d1, self.d2, self.d3, self.d4, self.d5, self.d6,
self.d7, self.d8]

        # Set initial logic states
        for i in range(1, 21):  # 20 pin IC
            self.logic_dict['c' + str(i)] = False

        self.filename = Path(__file__).parent /
"../images/ics/74273_easy_100x155.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
```

```
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        offset = self.offset

        # Note: Connector names correspond to pin-numbers, c1 = pin 1
        # Left side connectors
        self.conn_list.append(Connector(self.canvas, "c1", center.x - w / 2,
center.y - h / 2 +
                                        offset + conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "c3", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c4", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "c7", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 4))
        self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 5))
        self.conn_list.append(Connector(self.canvas, "c13", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 6))
        self.conn_list.append(Connector(self.canvas, "c14", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 7))
        self.conn_list.append(Connector(self.canvas, "c17", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 8))
        self.conn_list.append(Connector(self.canvas, "c18", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 9))
        self.conn_list.append(Connector(self.canvas, "c10", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 10))
```

```
        # Right side connectors
        self.conn_list.append(Connector(self.canvas, "c20", center.x + w / 2,
center.y - h/2 + offset +
                                        conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "c2", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c5", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "c6", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 4))
        self.conn_list.append(Connector(self.canvas, "c9", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 5))
        self.conn_list.append(Connector(self.canvas, "c12", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 6))
        self.conn_list.append(Connector(self.canvas, "c15", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 7))
        self.conn_list.append(Connector(self.canvas, "c16", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 8))
        self.conn_list.append(Connector(self.canvas, "c19", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 9))
        self.conn_list.append(Connector(self.canvas, "c11", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 10))

    def update_connectors(self):
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        offset = self.offset

        # Update left side pins
        for i in range(0, 10):
            self.conn_list[i].x, self.conn_list[i].y = (center.x - w / 2,
center.y - h / 2 +
                                                        offset + conn_inc *
```

```
(i+1))

        # Update right side pins
        for j in range(0, 10):
            self.conn_list[j+10].x, self.conn_list[j+10].y = (center.x + w / 2,
center.y - h / 2 +
                                                              offset + conn_inc
* (j+1))

        # Draw the connectors
        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            # print(wire.connector_obj.name)
            if wire.connector_obj.name == "c1":  # CLR input
                self.logic_dict['c1'] = wire.wire_obj.state
                if not wire.wire_obj.state:
                    for ff in self.ff_list:
                        ff.clear_ic()
                    self.set_q_outputs()
            elif wire.connector_obj.name == "c11":  # CLK input
                self.logic_dict['c9'] = wire.wire_obj.state
                if wire.wire_obj.state:
                    for ff in self.ff_list:
                        ff.Q1 = ff.D1
                    self.set_q_outputs()
            elif wire.connector_obj.name == "c2":  # 1Q output
                wire.wire_obj.state = self.logic_dict['c2']
            elif wire.connector_obj.name == "c3":  # 1D input
                self.d1.D1 = wire.wire_obj.state
                self.logic_dict['c3'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c4":  # 2D input
                self.d2.D1 = wire.wire_obj.state
                self.logic_dict['c4'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c5":  # 2Q Output
                wire.wire_obj.state = self.logic_dict['c5']
            elif wire.connector_obj.name == "c6":  # 3Q Output
                wire.wire_obj.state = self.logic_dict['c6']
            elif wire.connector_obj.name == "c7":  # 3D input
                self.d3.D1 = wire.wire_obj.state
                self.logic_dict['c7'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c8":  # 4D input
```

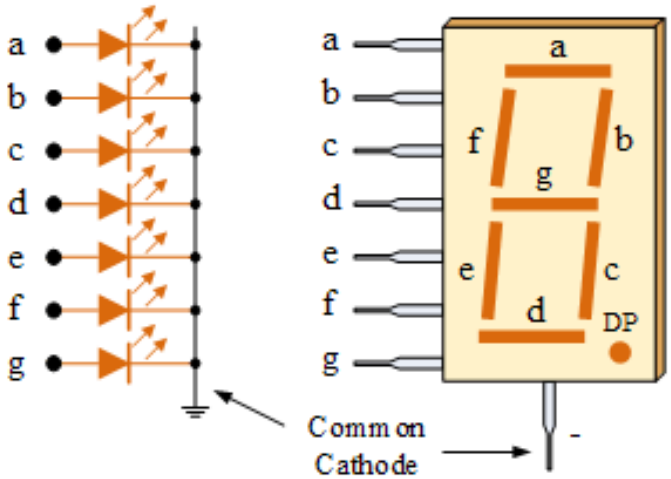
```
                self.d4.D1 = wire.wire_obj.state
                self.logic_dict['c8'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c9":  # 4Q output
                wire.wire_obj.state = self.logic_dict['c9']
            elif wire.connector_obj.name == "c10":  # GND - input
                self.logic_dict['c10'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c12":  # 5Q output
                wire.wire_obj.state = self.logic_dict['c12']
            elif wire.connector_obj.name == "c13":  # 5D input
                self.d5.D1 = wire.wire_obj.state
                self.logic_dict['c13'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c14":  # 6D input
                self.d6.D1 = wire.wire_obj.state
                self.logic_dict['c14'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c15":  # 6Q output
                wire.wire_obj.state = self.logic_dict['c15']
            elif wire.connector_obj.name == "c16":  # 7Q output
                wire.wire_obj.state = self.logic_dict['c16']
            elif wire.connector_obj.name == "c17":  # 7D input
                self.d7.D1 = wire.wire_obj.state
                self.logic_dict['c17'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c18":  # 8D input
                self.d8.D1 = wire.wire_obj.state
                self.logic_dict['c18'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c19":  # 8Q output
                wire.wire_obj.state = self.logic_dict['c19']
            elif wire.connector_obj.name == "c20":  # VCC input
                self.logic_dict['c20'] = wire.wire_obj.state

    def set_q_outputs(self):
        self.logic_dict['c2'] = self.d1.Q1  # 1Q
        self.logic_dict['c5'] = self.d2.Q1  # 2Q
        self.logic_dict['c6'] = self.d3.Q1  # 3Q
        self.logic_dict['c9'] = self.d4.Q1  # 4Q
        self.logic_dict['c12'] = self.d1.Q1  # 5Q
        self.logic_dict['c15'] = self.d2.Q1  # 6Q
        self.logic_dict['c16'] = self.d3.Q1  # 7Q
        self.logic_dict['c19'] = self.d4.Q1  # 8Q
```

7-Segment Display Class

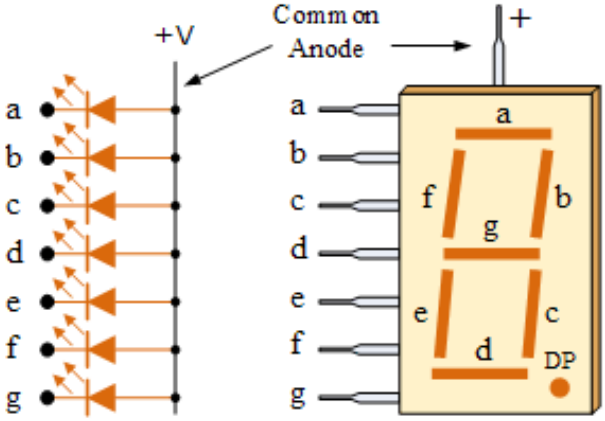
Objectives:

- Create a new 7-Segment display class
- Draw the display using shapes so that segment colors are programmable

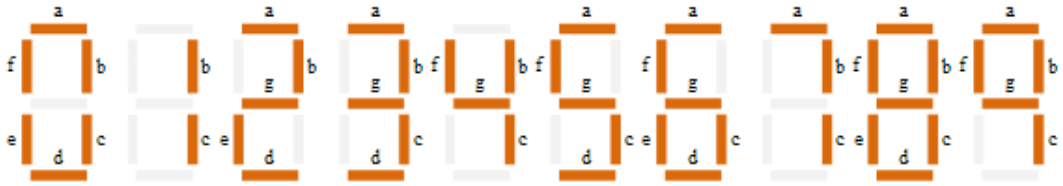
Common Cathode Configuration



Common Anode Configuration

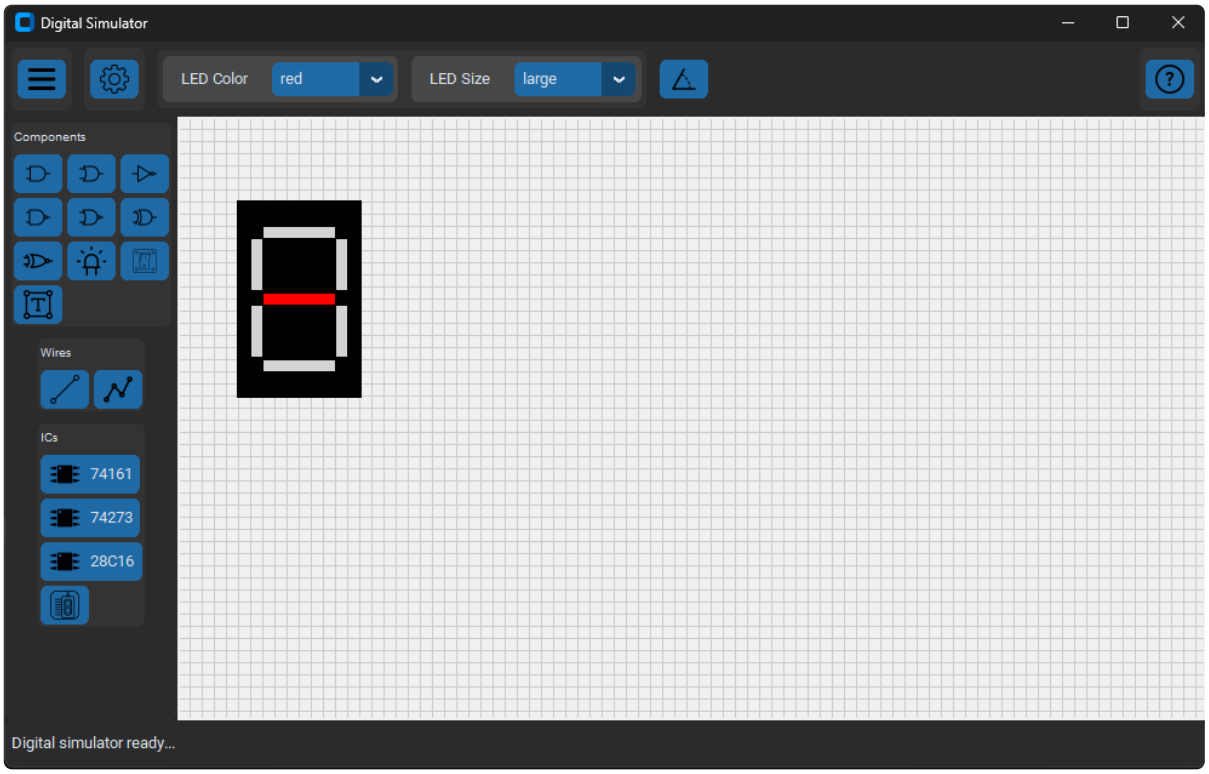


Segments needed for all Numbers



Truth Table

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	×	×	×	×	×	×	
1		×	×				
2	×	×		×	×		×
3	×	×	×	×			×
4		×	×			×	×
5	×		×	×		×	×
6	×		×	×	×	×	×
7	×	×	×				
8	×	×	×	×	×	×	×
9	×	×	×			×	×



IC_Lib/ic_7_segment_display.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class Segment:
    def __init__(self, canvas, x1, y1, orientation):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.orientation = orientation

        if self.orientation == "H":
            self.w = 60
            self.h = 10
        elif self.orientation == "V":
            self.w = 10
            self.h = 43

        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
```

```
        self.id = None
        self.state = False

        self.create_segment()

    def create_segment(self):
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2, fill="light gray")

    def update(self):
        self.update_position()
        self.update_color()

    def update_position(self):
        """Update the position when the object is moved"""
        self.canvas.coords(self.x1, self.y1, self.x2, self.y2)

    def update_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="red")
        else:
            self.canvas.itemconfig(self.id, fill="light gray")


class SevenSegment(IC):
    """Model for 7-Segment Display - 7-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "7_segment"

        # IC dimensions
        self.w = 100
        self.h = 160
        self.bbox = None

        # Initialize shape ids
        self.id = None
        self.seg_a, self.seg_b, self.seg_c, self.seg_d, self.seg_e, self.seg_f,
self.seg_g = (
            None, None, None, None, None, None, None)
        self.seg_list = []

        self.create_display()
        self.create_segments()
        self.update_bbox()
        self.create_selector()
```

```
    def create_display(self):
        # Assume black background with red segments
        self.id = self.canvas.create_rectangle(self.x1 - self.w/2, self.y1 -
self.h/2,
                                               self.x1 + self.w/2, self.y1 +
self.h/2,
                                               fill="black", outline="black",
width=3)

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_segments()
        self.update_selector()

    def update_position(self):
        """Update the position when the object is moved"""
        self.canvas.coords(self.x1 - self.w/2, self.y1 - self.h/2,
                           self.x1 + self.w/2, self.y1 + self.h/2)

    def create_segments(self):
        # Horizontal segments
        self.seg_a = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 -
self.h/2 + 20, "H")
        self.seg_list.append(self.seg_a)
        self.seg_g = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 - 5,
"H")
        self.seg_list.append(self.seg_g)
        self.seg_g.state = True
        self.seg_d = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 +
self.h/2 - 30, "H")
        self.seg_list.append(self.seg_d)

        # Vertical segments
        self.seg_f = Segment(self.canvas, self.x1 - self.w/2 + 10, self.y1 -
self.h/2 + 30, "V")
        self.seg_list.append(self.seg_f)
        self.seg_b = Segment(self.canvas, self.x1 + self.w/2 - 20, self.y1 -
self.h/2 + 30, "V")
        self.seg_list.append(self.seg_b)
        self.seg_e = Segment(self.canvas, self.x1 - self.w/2 + 10, self.y1 + 5,
"V")
        self.seg_list.append(self.seg_e)
        self.seg_c = Segment(self.canvas, self.x1 + self.w/2 - 20, self.y1 + 5,
"V")
```

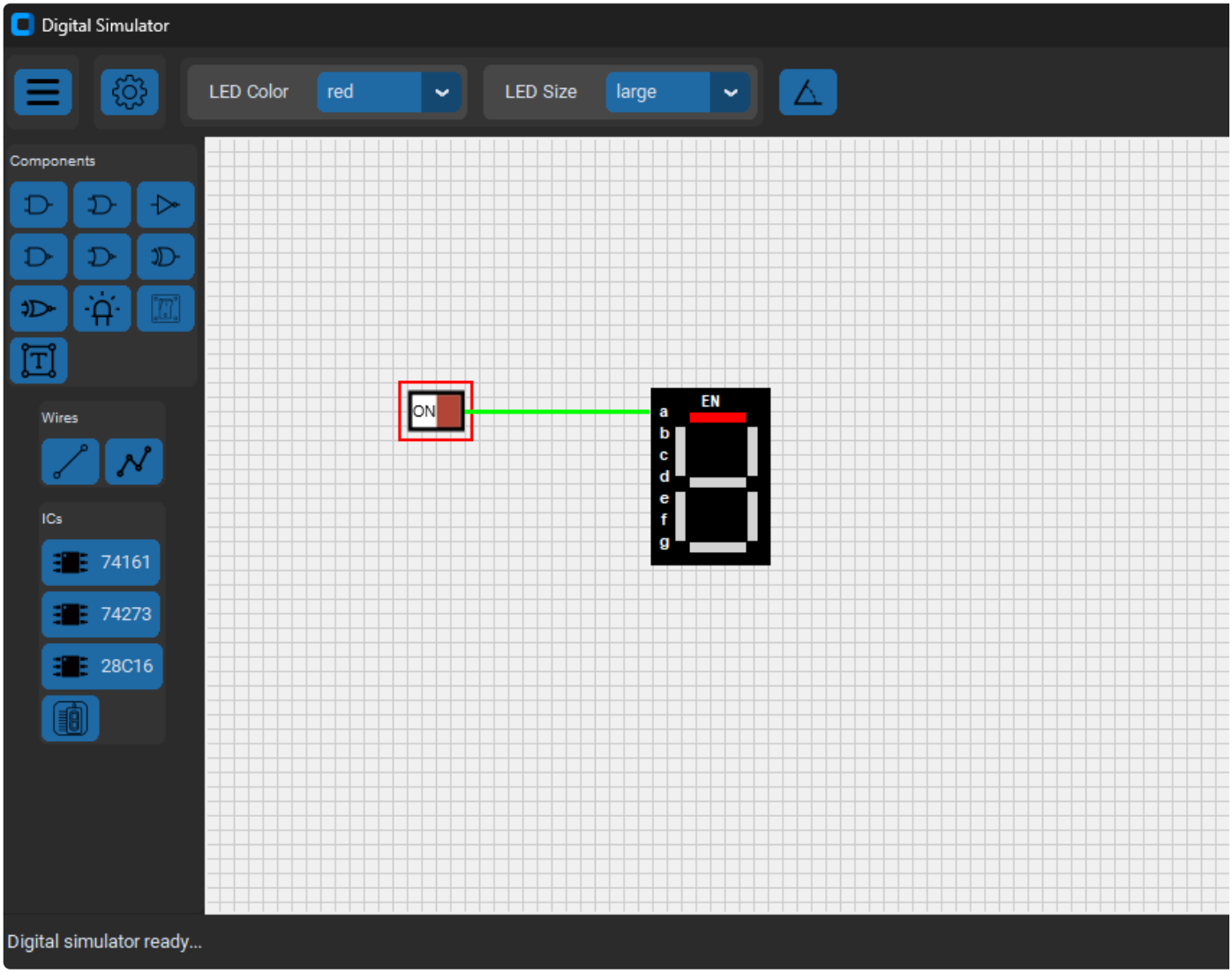
```
        self.seg_list.append(self.seg_c)


    def update_segments(self):
        for s in self.seg_list:
            s.update()

        # TODO: Add connectors and set_logic_level
```

UI_Lib/ic_button_frame.py

```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from IC_Lib import IC74161, IC74273, IC28C16, SevenSegment


class ICButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="ICs", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        ic_image = ctk.CTkImage(light_image=Image.open
                (Path(__file__).parent / "../icons/ic.png"),
               dark_image=Image.open
               (Path(__file__).parent / "../icons/ic.png"),
               size=(24, 24))

        ic_button = ctk.CTkButton(self, text="74161", image=ic_image, width=30,
                                          command=self.create_ic_74161)
        ic_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

        ic_button = ctk.CTkButton(self, text="74273", image=ic_image, width=30,
                                          command=self.create_ic_74273)
        ic_button.grid(row=2, column=0, sticky=ctk.W, padx=2, pady=2)
```

```
        ic_button = ctk.CTkButton(self, text="28C16", image=ic_image, width=30,
                                              command=self.create_ic_28C16)
        ic_button.grid(row=3, column=0, sticky=ctk.W, padx=2, pady=2)

        seven_segment_image = ctk.CTkImage(light_image=Image.open  # Added new
image
                (Path(__file__).parent / "../icons/7-segment-display.png"),
                dark_image=Image.open
                (Path(__file__).parent / "../icons/7-segment-display.png"),
                size=(24, 24))

        seven_segment_button = ctk.CTkButton(self, text="",
image=seven_segment_image, width=30,  # Added new button
                                     command=self.create_seven_segment)
        seven_segment_button.grid(row=4, column=0, sticky=ctk.W, padx=2,
pady=2)  # Added new button

    def create_ic_74161(self):
        ic = IC74161(self.canvas, 100, 100)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

    def create_ic_74273(self):
        ic = IC74273(self.canvas, 105, 100)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

    def create_ic_28C16(self):
        ic = IC28C16(self.canvas, 100, 150)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

    def create_seven_segment(self):  # Added new method
        ic = SevenSegment(self.canvas, 100, 150)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()
```

Connectors and Logic



IC_Lib/ic_7_segment_display.py

```
import customtkinter as ctk
from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector
from Comp_Lib import Text


class Segment:
    def __init__(self, canvas, x1, y1, orientation):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.orientation = orientation
```

```
        if self.orientation == "H":
            self.w = 40
            self.h = 8
        elif self.orientation == "V":
            self.w = 8
            self.h = 35

        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
        self.id = None
        self.state = False

        self.create_segment()

    def create_segment(self):
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2, fill="light gray")

    def update(self):
        self.update_position()
        self.update_color()

    def update_position(self):
        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
        """Update the position when the object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)

    def update_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="red")
        else:
            self.canvas.itemconfig(self.id, fill="light gray")


class SevenSegment(IC):
    """Model for 7-Segment Display - 7-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "7_segment"

        # IC dimensions
        self.w = 80
        self.h = 120
        self.bbox = None
```

```
        self.conn_inc = 15
        self.label_list = []
        self.logic_dict = {'a': False, 'b': False, 'c': False, 'd': False, 'e':
False, 'f': False, 'g': False,
                           'EN': False}

        # Initialize shape ids
        self.id = None
        self.seg_a, self.seg_b, self.seg_c, self.seg_d, self.seg_e, self.seg_f,
self.seg_g = (
            None, None, None, None, None, None, None)
        self.seg_list = []

        self.create_display()
        self.create_segments()
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()
        self.create_labels()

    def create_display(self):
        # Assume black background with red segments
        self.id = self.canvas.create_rectangle(self.x1 - self.w/2, self.y1 -
self.h/2,
                                               self.x1 + self.w/2, self.y1 +
self.h/2,
                                               fill="black", outline="black",
width=3, tags='display')

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_bbox()
        self.update_segments()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()
        self.update_labels()

    def update_position(self):
        """Update the position when the object is moved"""
        self.canvas.coords(self.id, self.x1 - self.w/2, self.y1 - self.h/2,
                           self.x1 + self.w/2, self.y1 + self.h/2)
```

```
    def create_segments(self):
        # Horizontal segments
        self.seg_a = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1 -
self.h/2 + 15, "H")
        self.seg_list.append(self.seg_a)
        self.seg_g = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1,
"H")
        self.seg_list.append(self.seg_g)
        self.seg_d = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1 +
self.h/2 - 15, "H")
        self.seg_list.append(self.seg_d)

        # Vertical segments
        self.seg_f = Segment(self.canvas, self.x1 - self.w/2 + 15, self.y1 -
self.h/2 + 25, "V")
        self.seg_list.append(self.seg_f)
        self.seg_b = Segment(self.canvas, self.x1 + self.w/2 - 25, self.y1 -
self.h/2 + 25, "V")
        self.seg_list.append(self.seg_b)
        self.seg_e = Segment(self.canvas, self.x1 - self.w/2 + 15, self.y1 +
10, "V")
        self.seg_list.append(self.seg_e)
        self.seg_c = Segment(self.canvas, self.x1 + self.w/2 - 25, self.y1 +
10, "V")
        self.seg_list.append(self.seg_c)

    def update_segments(self):
        self.seg_a.x1, self.seg_a.y1 = self.x1 - self.w/2 + 25, self.y1 -
self.h/2 + 15
        self.seg_b.x1, self.seg_b.y1 = self.x1 + self.w/2 - 15, self.y1 -
self.h/2 + 25
        self.seg_c.x1, self.seg_c.y1 = self.x1 + self.w/2 - 15, self.y1 + 10
        self.seg_d.x1, self.seg_d.y1 = self.x1 - self.w/2 + 25, self.y1 +
self.h/2 - 15
        self.seg_e.x1, self.seg_e.y1 = self.x1 - self.w/2 + 15, self.y1 + 10
        self.seg_f.x1, self.seg_f.y1 = self.x1 - self.w/2 + 15, self.y1 -
self.h/2 + 25
        self.seg_g.x1, self.seg_g.y1 = self.x1 - self.w/2 + 25, self.y1
        for s in self.seg_list:
            s.update()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
```

```
        conn_inc = self.conn_inc

        # Left side connectors
        self.conn_list.append(Connector(self.canvas, "a", center.x - w / 2,
center.y - conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "b", center.x - w / 2,
center.y - conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c", center.x - w / 2,
center.y - conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "d", center.x - w / 2,
center.y))
        self.conn_list.append(Connector(self.canvas, "e", center.x - w / 2,
center.y + conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "f", center.x - w / 2,
center.y + conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "g", center.x - w / 2,
center.y + conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "EN", center.x, center.y -
h / 2))

    def update_connectors(self):
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc

        self.conn_list[0].x, self.conn_list[0].y = center.x - w / 2, center.y -
conn_inc * 3
        self.conn_list[1].x, self.conn_list[1].y = center.x - w / 2, center.y -
conn_inc * 2
        self.conn_list[2].x, self.conn_list[2].y = center.x - w / 2, center.y -
conn_inc * 1
        self.conn_list[3].x, self.conn_list[3].y = center.x - w / 2, center.y
        self.conn_list[4].x, self.conn_list[4].y = center.x - w / 2, center.y +
conn_inc * 1
        self.conn_list[5].x, self.conn_list[5].y = center.x - w / 2, center.y +
conn_inc * 2
        self.conn_list[6].x, self.conn_list[6].y = center.x - w / 2, center.y +
conn_inc * 3

        self.conn_list[7].x, self.conn_list[7].y = center.x, center.y - h / 2

        # Draw the connectors
        for c in self.conn_list:
```

```
            c.update()

        self.move_connected_wires()

    def create_labels(self):
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        pos = 10

        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 3, text='a', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 2, text='b', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 1, text='c', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y, text='d', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 1, text='e', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 2, text='f', fill="white"))
        self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 3, text='g', fill="white"))
        self.label_list.append(Text(self.canvas, center.x, center.y - h / 2  +
pos, text='EN', fill="white"))

    def update_labels(self):
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        pos = 10

        self.label_list[0].x1, self.label_list[0].y1 = center.x - w / 2 + pos,
center.y - conn_inc * 3
        self.label_list[1].x1, self.label_list[1].y1 = center.x - w / 2 + pos,
center.y - conn_inc * 2
        self.label_list[2].x1, self.label_list[2].y1 = center.x - w / 2 + pos,
center.y - conn_inc * 1
        self.label_list[3].x1, self.label_list[3].y1 = center.x - w / 2 + pos,
center.y
        self.label_list[4].x1, self.label_list[4].y1 = center.x - w / 2 + pos,
```

```
center.y + conn_inc * 1
        self.label_list[5].x1, self.label_list[5].y1 = center.x - w / 2 + pos,
center.y + conn_inc * 2
        self.label_list[6].x1, self.label_list[6].y1 = center.x - w / 2 + pos,
center.y + conn_inc * 3
        self.label_list[7].x1, self.label_list[7].y1 = center.x, center.y - h /
2  + pos

        for label in self.label_list:
            self.canvas.coords(label.id, label.x1, label.y1)

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "a":
                self.seg_a.state = wire.wire_obj.state
            elif wire.connector_obj.name == "b":
                self.seg_b.state = wire.wire_obj.state
            elif wire.connector_obj.name == "c":
                self.seg_c.state = wire.wire_obj.state
            elif wire.connector_obj.name == "d":
                self.seg_d.state = wire.wire_obj.state
            elif wire.connector_obj.name == "e":
                self.seg_e.state = wire.wire_obj.state
            elif wire.connector_obj.name == "f":
                self.seg_f.state = wire.wire_obj.state
            elif wire.connector_obj.name == "g":
                self.seg_g.state = wire.wire_obj.state
            elif wire.connector_obj.name == "EN":
                pass
```

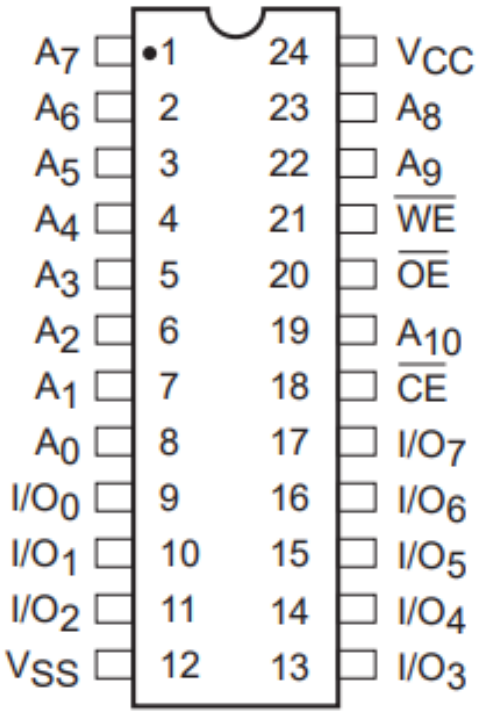
28C16 2K x 8-Bit EEPROM Class

Objectives:

- Create a 28C16 EEPROM class
- Create a "program" for converting binary input to decimal output for 7-segment display
- Program to support 4-bit conversion from 0x00 - 0xFF hex to 0 - 15 dec
- Binary input on address pins A0 to A10 - 11-bit address = 2048 addresses

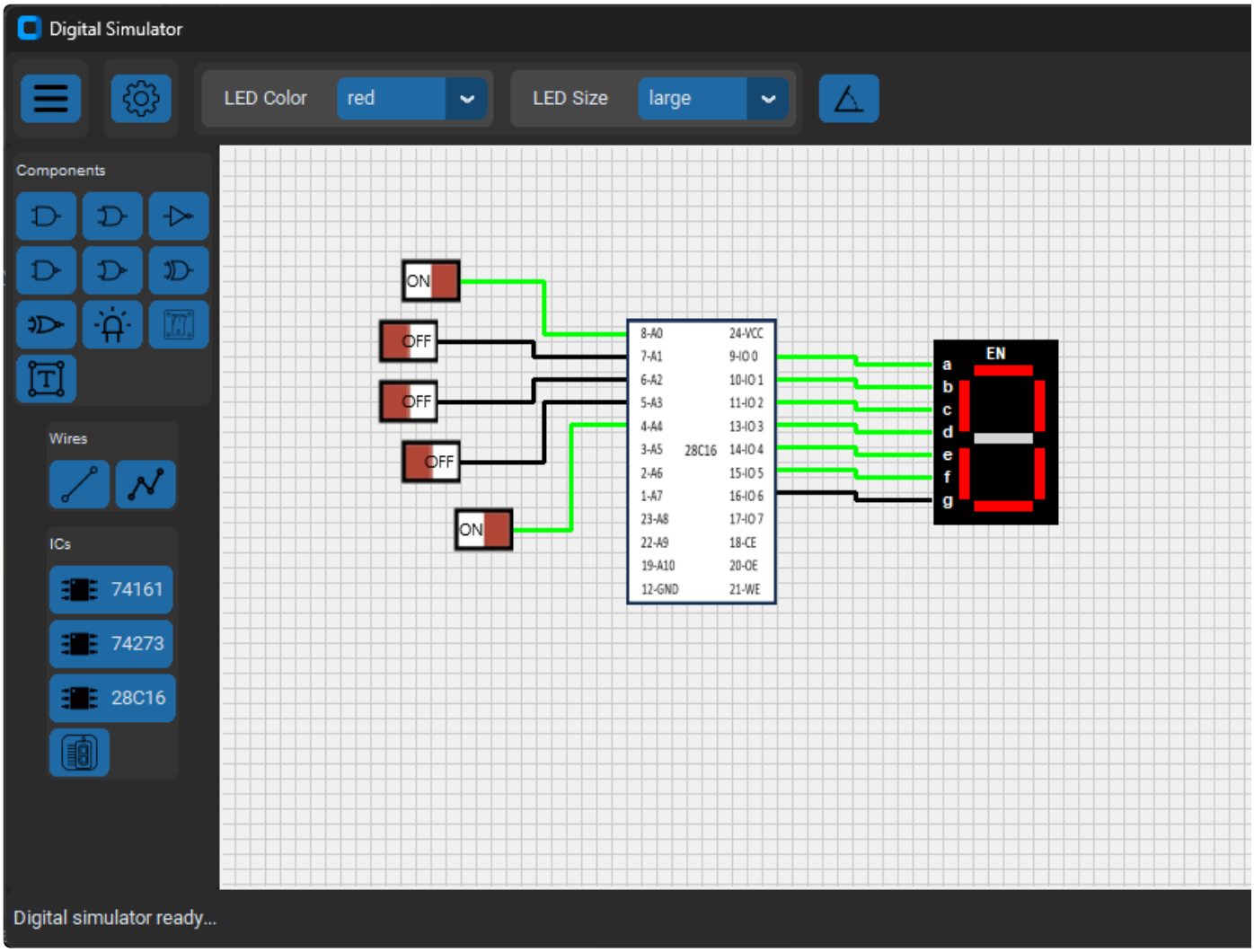
28C16 Data Sheet

Device pin-out



Device pin functions

Pin Name	Function
A_0–A_{10}	Address Inputs
I/O_0–I/O_7	Data Inputs/Outputs
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
$\overline{WE}$	Write Enable
V_{CC}	5V Supply
V_{SS}	Ground
NC	No Connect



IC_Lib/ic_28C16_eeprom.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class IC28C16(IC):
    """Model for 28C16 2K x 8-Bit EEPROM - 24-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic28C16"

        self.logic_dict = {}
        self.conn_inc = 15
        self.offset = -5
```

```
        # Set initial logic states
        for i in range(1, 25):  # 24 pin IC
            self.logic_dict['c' + str(i)] = False

        self.zero, self.one, self.two, self.three, self.four, self.five,
self.six, self.seven, self.eight, self.nine = (
            None, None, None, None, None, None, None, None, None, None)
        self.set_display_data()

        self.filename = Path(__file__).parent /
"../images/ics/28C16_easy_100x190.png"
        self.create_ic()

        # Define program for Display 0-1
        self.program1 = {
            '0000': self.zero,
            '0001': self.zero,
            '0010': self.zero,
            '0011': self.zero,
            '0100': self.zero,
            '0101': self.zero,
            '0110': self.zero,
            '0111': self.zero,
            '1000': self.zero,
            '1001': self.zero,
            '1010': self.one,
            '1011': self.one,
            '1100': self.one,
            '1101': self.one,
            '1110': self.one,
            '1111': self.one
        }

        # Define program for Display 0-9
        self.program2 = {
            '0000': self.zero,
            '0001': self.one,
            '0010': self.two,
            '0011': self.three,
            '0100': self.four,
            '0101': self.five,
            '0110': self.six,
            '0111': self.seven,
            '1000': self.eight,
            '1001': self.nine,
            '1010': self.zero,
```

```
            '1011': self.one,
            '1100': self.two,
            '1101': self.three,
            '1110': self.four,
            '1111': self.five
        }

    def set_display_data(self):
        # Set output for a, b, c, d, e, f, g on the display
        self.zero = [True, True, True, True, True, True, False]
        self.one = [False, True, True, False, False, False, False]
        self.two = [True, True, False, True, True, False, True]
        self.three = [True, True, True, True, False, False, True]
        self.four = [False, True, True, False, False, True, True]
        self.five = [True, False, True, True, False, True, True]
        self.six = [True, False, True, True, True, True, True]
        self.seven = [True, True, True, False, False, False, False]
        self.eight = [True, True, True, True, True, True, True]
        self.nine = [True, True, True, False, False, True, True]

    def create_ic(self):
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        offset = self.offset

        # Note: Connector names correspond to pin-numbers, c1 = pin 1
        # Left side connectors
```

```
        self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 +
                                        offset + conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "c7", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c6", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "c5", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 4))
        self.conn_list.append(Connector(self.canvas, "c4", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 5))
        self.conn_list.append(Connector(self.canvas, "c3", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 6))
        self.conn_list.append(Connector(self.canvas, "c2", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 7))
        self.conn_list.append(Connector(self.canvas, "c1", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 8))
        self.conn_list.append(Connector(self.canvas, "c23", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 9))
        self.conn_list.append(Connector(self.canvas, "c22", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 10))
        self.conn_list.append(Connector(self.canvas, "c19", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 11))
        self.conn_list.append(Connector(self.canvas, "c12", center.x - w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 12))

        # Right side connectors
        self.conn_list.append(Connector(self.canvas, "c24", center.x + w / 2,
center.y - h/2 + offset +
                                        conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "c9", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c10", center.x + w / 2,
center.y - h / 2 + offset +
```

```
                                        conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "c11", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 4))
        self.conn_list.append(Connector(self.canvas, "c13", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 5))
        self.conn_list.append(Connector(self.canvas, "c14", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 6))
        self.conn_list.append(Connector(self.canvas, "c15", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 7))
        self.conn_list.append(Connector(self.canvas, "c16", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 8))
        self.conn_list.append(Connector(self.canvas, "c17", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 9))
        self.conn_list.append(Connector(self.canvas, "c18", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 10))
        self.conn_list.append(Connector(self.canvas, "c20", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 11))
        self.conn_list.append(Connector(self.canvas, "c21", center.x + w / 2,
center.y - h / 2 + offset +
                                        conn_inc * 12))

    def update_connectors(self):
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        offset = self.offset

        # Update left side pins
        for i in range(0, 12):
            self.conn_list[i].x, self.conn_list[i].y = (center.x - w / 2,
center.y - h / 2 +
                                                        offset + conn_inc *
(i+1))

        # Update right side pins
```

```
        for j in range(0, 12):
            self.conn_list[j+12].x, self.conn_list[j+12].y = (center.x + w / 2,
center.y - h / 2 +
                                                              offset + conn_inc
* (j+1))

        # Draw the connectors
        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "c1":
                self.logic_dict['c1'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c2":
                self.logic_dict['c2'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c3":
                self.logic_dict['c3'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c4":
                self.logic_dict['c4'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c5":
                self.logic_dict['c5'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c6":
                self.logic_dict['c6'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c7":
                self.logic_dict['c7'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c8":
                self.logic_dict['c8'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c9":  # I/O 0
                wire.wire_obj.state = self.logic_dict['c9']
            elif wire.connector_obj.name == "c10":  # I/O 1
                wire.wire_obj.state = self.logic_dict['c10']
            elif wire.connector_obj.name == "c11":  # I/O 2
                wire.wire_obj.state = self.logic_dict['c11']
            elif wire.connector_obj.name == "c12":  # VSS = GND
                self.logic_dict['c12'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c13":  # I/O 3
                wire.wire_obj.state = self.logic_dict['c13']
            elif wire.connector_obj.name == "c14":  # I/O 4
                wire.wire_obj.state = self.logic_dict['c14']
            elif wire.connector_obj.name == "c15":  # I/O 5
                wire.wire_obj.state = self.logic_dict['c15']
            elif wire.connector_obj.name == "c16":  # I/O 6
                wire.wire_obj.state = self.logic_dict['c16']
```

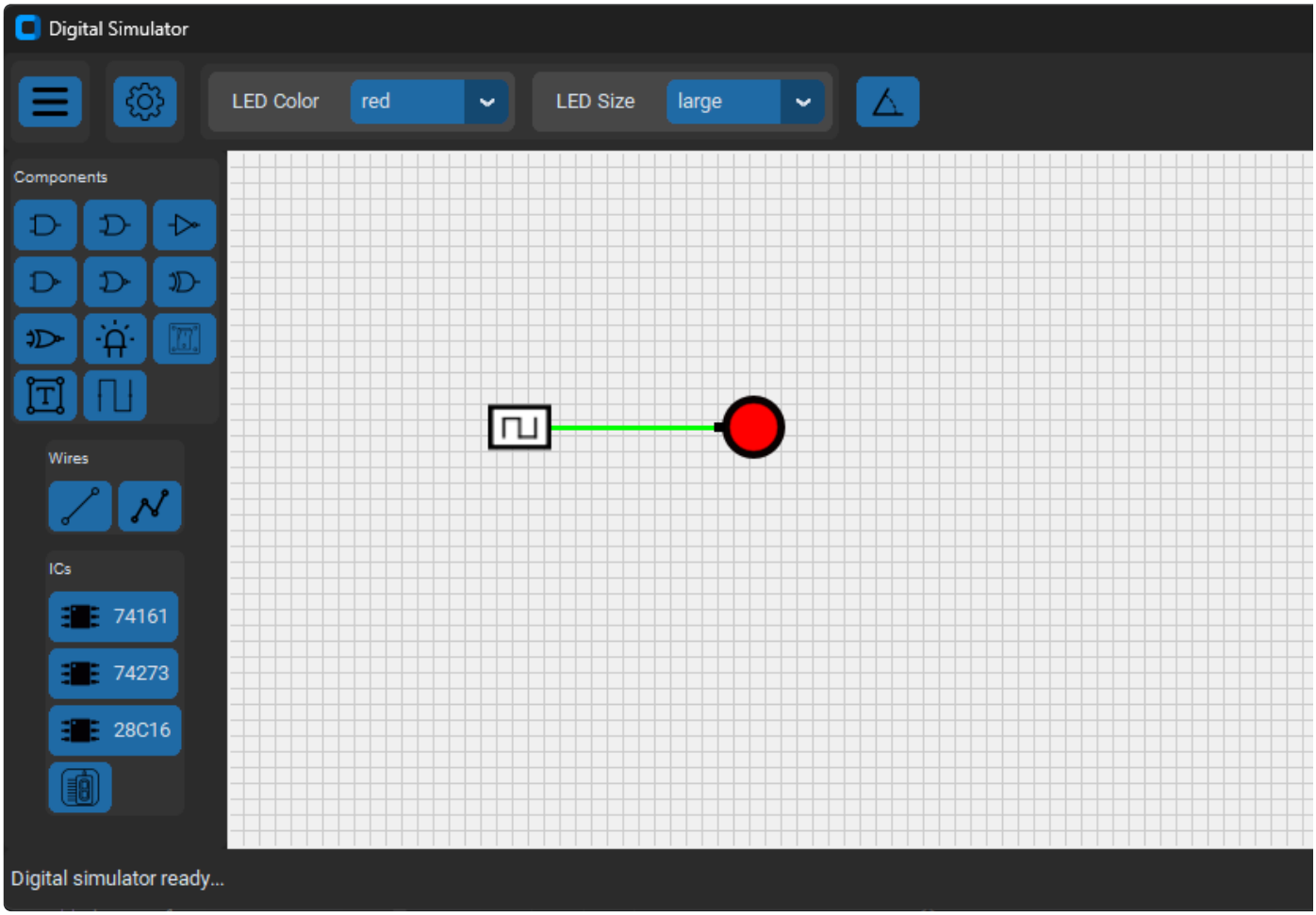
```
            elif wire.connector_obj.name == "c17":  # I/O 7
                wire.wire_obj.state = self.logic_dict['c17']
            elif wire.connector_obj.name == "c18":  # !CE
                self.logic_dict['c18'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c19":  # A10
                self.logic_dict['c19'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c20":  # !OE
                self.logic_dict['c20'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c21":  # !WE
                self.logic_dict['c21'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c22":  # A9
                self.logic_dict['c22'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c23":  # A8
                self.logic_dict['c23'] = wire.wire_obj.state
            elif wire.connector_obj.name == "c24":  # VCC
                self.logic_dict['c24'] = wire.wire_obj.state

        key, ps = self.convert_address_to_key()
        if ps is True:  # Select program #1
            io = self.program1[key]
        else:  # Select program #2
            io = self.program2[key]

        self.set_io_output(io)

    def convert_address_to_key(self):
        a0 = self.logic_dict['c8']  # Counter bit 0
        a1 = self.logic_dict['c7']  # Counter bit 1
        a2 = self.logic_dict['c6']  # Counter bit 2
        a3 = self.logic_dict['c5']  # Counter bit 3

        ps = self.logic_dict['c4']  # EEPROM Program Select

        result = lambda s: '1' if s is True else '0'
        k0 = result(a0)
        k1 = result(a1)
        k2 = result(a2)
        k3 = result(a3)
        key = k3 + k2 + k1 + k0
        return key, ps

    def set_io_output(self, io):
        self.logic_dict['c9']  = io[0]  # i/o 0 = a
        self.logic_dict['c10']  = io[1]  # i/o 1 = b
        self.logic_dict['c11']  = io[2]  # i/o 2 = c
        self.logic_dict['c13']  = io[3]  # i/o 3 = d
```

```
        self.logic_dict['c14']  = io[4]  # i/o 4 = e
        self.logic_dict['c15']  = io[5]  # i/o 5 = f
        self.logic_dict['c16']  = io[6]  # i/o 6 = g
        self.logic_dict['c17']  = False  # Not Used
```

Clock Class

Objectives:

- Create a new clock class
- Create the logic on a separate thread to improve performance



Comp_Lib/clock.py

```
import threading
import time
```

```
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector
from Helper_Lib.point import Point


class Clock(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "clock"
        self.out_state = False  # OFF state

        self.filename =
"D:/EETools/DigitalSimulator/images/switch/clock_40x30.png"

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 1 connector
        self.out1_id = None
        self.create_connectors()
        self.set_connector_visibility()

        # Start the clock
        self.thread = threading.Thread(target=self.toggle_clock)
        self.thread.start()

    def __del__(self):
        print('Clock destructor called')
        self.thread.join()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):  # Added new method
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
```

```
        # Define 1 connector: out1
        self.out1_id = Connector(self.canvas, "out", center.x + w / 2,
center.y)
        self.conn_list = [self.out1_id]
        self.set_connector_visibility()

    def update_connectors(self):  # Added new method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Update connector position based on angle
        if self.angle == 0:
            self.out1_id.x, self.out1_id.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.out1_id.x, self.out1_id.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.out1_id.x, self.out1_id.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.out1_id.x, self.out1_id.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def toggle_clock(self):
        # set the time
        time.sleep(1)

        # Toggle state
        self.out_state = not self.out_state

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
            self.canvas.redraw_no_grid()

        self.toggle_clock()
```

UI_Lib/canvas.py

```
. . .

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

    def redraw_no_grid(self):  # Added new method
        for c in self.comp_list:
            c.update()

. . .
```

UI_Lib/comp_button_frame.py

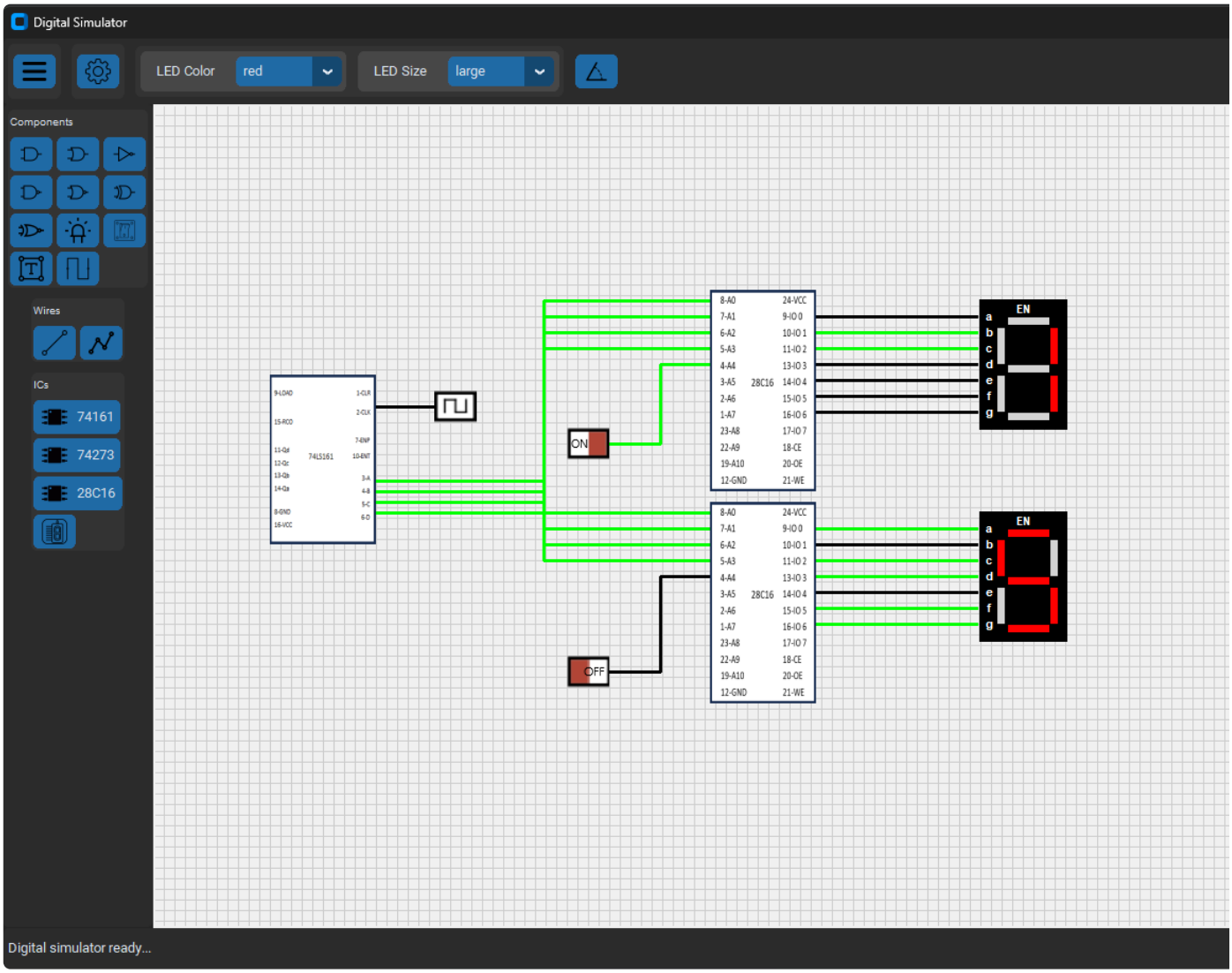
```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import AndGate, OrGate, NotGate, NandGate, NorGate, XorGate,
XnorGate
from Comp_Lib import LED, Switch, Text, Clock  # Added import for Clock Class


class CompButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("and_gate", "../icons/and.png"),
                            ("or_gate", "../icons/or.png"),
                            ("not_gate", "../icons/not.png"),
                            ("nand_gate", "../icons/nand.png"),
                            ("nor_gate", "../icons/nor.png"),
                            ("xor_gate", "../icons/xor.png"),
                            ("xnor_gate", "../icons/xnor.png"),
                            ("led", "../icons/led.png"),
                            ("switch", "../icons/switch.png"),
                            ("text", "../icons/text.png"),
```

```
                            ("clock", "../icons/clock.png")]  # Added icon for
clock

        self.init_frame_widgets()

. . .

        elif name == "text":
            gate = Text(self.canvas, 100, 100)
        elif name == "clock":  # Added clock class instatiation here
            gate = Clock(self.canvas, 100, 100)
        self.canvas.comp_list.append(gate)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()
```

Counter Circuit Simulation

Objective:

- Create a counter circuit
- 74161 Counter is driven by a Clock
- Add two 28C16 EEPROM
- Add two 7-segment displays
- Verify that the circuit counts from 0 to 15 and displays the count on the output displays
- Save and load the circuit from the file menu



Summary

This concludes the Digital Circuit Simulator advanced project development. The counter circuit simulation is pretty cool and we have developed all the modules to make it work. Python is definitely capable of complex circuit simulation. We will proceed to the next advanced project and my personal favorite - Microwave Circuit Simulation.