Pulchowk Campus
Institute Of Engineering
Tribhuvan University

C Project Report on

MinimaLogic

A Digital Logic Simulator

Submitted To:

Department of Electronics and Computer Engineering

Submitted By:

Aditi Kharel – PUL077BEI008

Ashutosh Bohara – PUL077BEI012

Pallavi Paudel – PUL077BEI027

Rijan Shrestha – PUL077BEI034

Acknowledgements

First of all, we would like to express our gratitude to our teacher, Anku Jaiswal, for her constant support and guidance throughout the semester. The lectures, assignments and guidelines have been of great help for us to successfully complete our project.

We would also like to thank IOE, Pulchowk campus and Department of Computer and Electronics Engineering for providing us an opportunity to work on this project. We have enhanced our programming skills in C language, and learnt to work as a team in the past few weeks. Our friends and seniors have also been incredibly helpful in figuring out this whole "teamwork for a project" scenario by sharing their knowledge and critique, this helped in shaping the outline and rectifying the elements of our project. We feel immensely thankful to be surrounded by such supportive, kind and obliging people. Our families have also played a huge role by providing us the support we needed-both emotional and mental-without which finishing the project would have been impossible. Everyone has a lot on one's plate, thus, we are thankful to everyone who took their time and helped us make the most of the opportunity we were handed with. We would like to sincerely thank everyone who has directly or indirectly contributed to the project.

Table of Contents

Chapter 1

INTRODUCTION

Logic circuits are one of the core components of CPUs. These circuits allow manipulating binary data and carrying out various logical operations. Programs such as Proteus and Logisim can be used to simulate logic circuits. For our project, we decided to create a similar (albeit heavily simplified) logic simulator named **MinimaLogic**.

MinimaLogic is a GUI based logic simulator of x64 Windows systems that allows simulation of various logic gates and circuits. It allows users to create and simulate circuits ranging from a simple 1-bit adder to more complex circuits like 4-bit counters. In fact, one can create any circuit that fits within the available area using the components provided in the program. The program allows the user to interact via mouse and keyboard. To make the program user friendly, we tried to keep the controls as intuitive as possible.

This program heavily relies on the Simple DirectMedia Layer(SDL2) Library for GUI elements as well as user interation/input. Alongside SDL2, the program also utilizes SDL2_ttf for font rendering as well as the Windows API for opening/saving files. Other standard headers that are available with all modern development environments have also been used.

The complete source code for the project can be found here:
https://github.com/First-Sem-C-Project/Minimalogic

1.1 The C Programming Language

C is a general-purpose, procedural computer programming language which was developed by Dennis Ritchie in Bell Laboratories between 1972 and 1973 AD as a successor to Basic Combined Programming Language (BCPL or the B language). C was designed such that code written in C can be translated efficiently into machine level instructions. Code

written in C somewhat resembles the english language as it uses keywords such as `if, else, for, do`, etc. Along with its speed and syntax, C contains addidtional features which allow it to work at lower level thus it can bridge the gap between machine level and high level languages. Due to this, C has found lasting use in systems programming (e.g writing operating systems). It can also be used for applications programming. Applications made in C are generally much faster and efficient than most other programming languages even with less optimization.

Some characteristics of the C language:

- The language has a small, fixed number of keywords (only 32), including a full set of control flow primitives: if/else, for, do/while, while, and switch.
- It has a rich set of operators (arithmetic, bitwise, relational, logical and some miscellaneous operators).
- It allows users to define functions that return value of certain data type however, value returned by function can be ignored if not needed.
- It also allows for procedures i.e. functions not returning values, by using a return type `void`.
- It is a statically typed language i.e. all data has a type, but implicit conversions are possible.
- Declaration syntax mimics usage context. For eg: C has no "define" keyword; instead, a statement beginning with the name of a type is taken as a declaration.
- It allows for user-defined (`typedef`) and compound data types through `struct`(structure, heterogeneous aggregate data type), `union`(structure with overlapping members), `enum`(enumerated data type) and arrays(homogeneous aggregate of data).
- It allows low-level access to computer memory by converting machine addresses to typed pointers.
- A preprocessor performs macro definition, source code file inclusion, and conditional compilation.
- There is a basic form of modularity: files can be compiled separately and linked together, with control over which functions and data objects are visible to other files via `static` and `extern` attributes.
- The standard library of C provides a rich set of functions which allow for complex functionality such as I/O, string manipulation, and mathematical functions.

1.2 Structure of Code

Generally, a C program is divided into following sections:

- Inclusion / Linking Section
  This section consists of header files to be included in the program. Inclusion of files is done using the `#include` preprocessor directive. It provides instructions to the compiler to link functions, structures, enums etc. from the header file.

- Macro Definition Section
  This sections consists of macro (or symbolic constants) definitions. Macros are defined using the `#define` preprocessor directive.

- Function Definition Section
  This section consists of definitions of all user defined funtions that are to be used in the program to perform. These functions are called from the `main()` function.

- Main Function Section
  There can only be one `main()` function in the entire C program. The `main()` function acts as the entry point to the program i.e. execution of program begins from the main function.

1.2.1 Sample C Program

```c
// Linking section
#include <stdio.h>
// Here, the C standard library stdio.h (standard input output) has been
    included

//Macro definition section
#define MSG "Hello, Peter\n"
//Here, a macro MSG has been defined which expands to "Hello, Peter\n"

//Function definition section
void PrintMessage(){
    printf(MSG);
}
//Here, a function PrintMessage() has been defined
//It calls the printf() function from stdio.h to display value of MSG to the
    screen

//Main Function Section
```

```
17 int main(){
18     PrintMessage();
19     return 0;
20 }
21 //Here, the main function has been defined which calls the PrintMessage()
       function defined previously.
22
23 // Output
24 // Hello, Peter
```

The source code for our program has been divided into 9 files (4 headers and 5 C files). Our source code mostly follows this format however at certain parts of the code the format has not been followed for the sake of convinience.

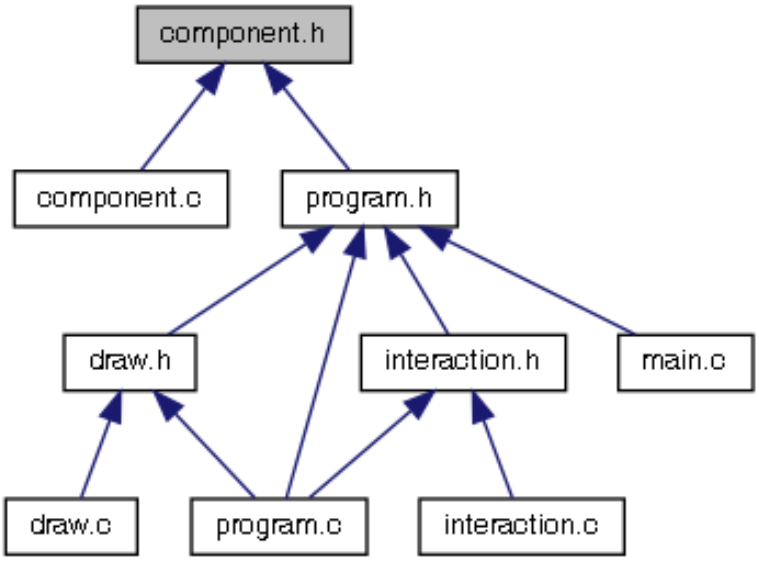

Figure 1.1: File inclusion graph

The files are linked as shown in figure above. Functions, structs and enums shared across multiple files have been defined in the relevant header file.

1.3 Simple DirectMedia Layer(SDL2)

Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games including Valve's award winning catalog and many Humble Bundle games.

SDL2 libraries also contains extension to keep SDL as light as possible. Some of the libraries are SDL2_image, SDL2_net, SDL2_mixer, SDL2_ttf, true type SDL2_rtf.

SDL2_image is used to load different images, SDL2_net is used for cross platform networking, SDL2_mixer is an audio mixer library that supports WAV, MP3, MIDI and OGG. SDL2_ttf is used to write using fonts in the program and SDL2_rtf is Rich Text Format library. Out of these libraries, we only used SDL2_ttf.

The SDL2 library is more oriented towards game development and it provides a vast quantity of features such as 3D rendering, hardware accelerated 2D graphics, multiple windows, multiple audio devices, multiple input devices etc. Since our program is much simpler than a full-fledged game, we were only able to utilize a small fraction of SDL's features.

SDL programs run continuously in a loop which is often referred to as the game loop. The loop can be summarized by the following diagram:

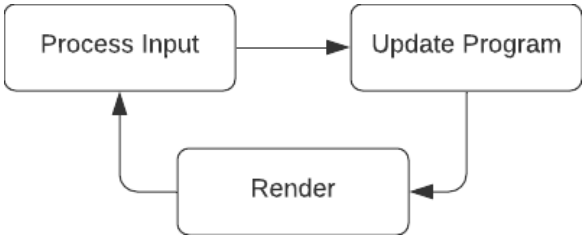

Figure 1.2: SDL program loop

1.3.1 Sample SDL Program

```
1  // This programs opens a SDL window which shows a blank screen until the user
       quits
2  #include <SDL2/SDL.h>
3  #include <stdio.h>
4
5  // Declare SDL window and renderer
6  SDL_Window *window;
7  SDL_Renderer *renderer;
8
9  int main(int argc, char ** argv) {
10     // Display error and quit program if SDL cannot be initialized
11     if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
12         printf("Failed to initialize SDL\n");
13         return -1;
14     }
15     // Create a window
16     window = SDL_CreateWindow("Demo", 0, 0, 500, 500, SDL_WINDOW_RESIZABLE);
```

```
    // Display error and quit program if window cannot be created
    if (!window) {
        printf("Could not create a window: %s", SDL_GetError());
        return -1;
    }
    // Create a renderer
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_SOFTWARE);
    // Display error and quit program if renderer cannot be created
    if (!renderer) {
        printf("Could not create a renderer: %s", SDL_GetError());
        return -1;
    }
    // Main program loop
    while (true) {
        // Get the next event
        SDL_Event event;
        if (SDL_WaitEventTimeout(&event, 10)) // Wait for event (user input) {
            if (event.type == SDL_QUIT) {
                // Exit loop if user presses quit
                break;
            }
                /* User Input Processing happens here */
        }
        // Clear the screen
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderClear(renderer);

        /* Rendering/drawing code goes here */
        /* Updating code goes here */

        SDL_RenderPresent(renderer); // Update the screen
        SDL_Delay(10); // Limit frame rate to reduce CPU load
    }
    // Clean up
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Chapter 2

METHODOLOGY

2.1 Flowchart

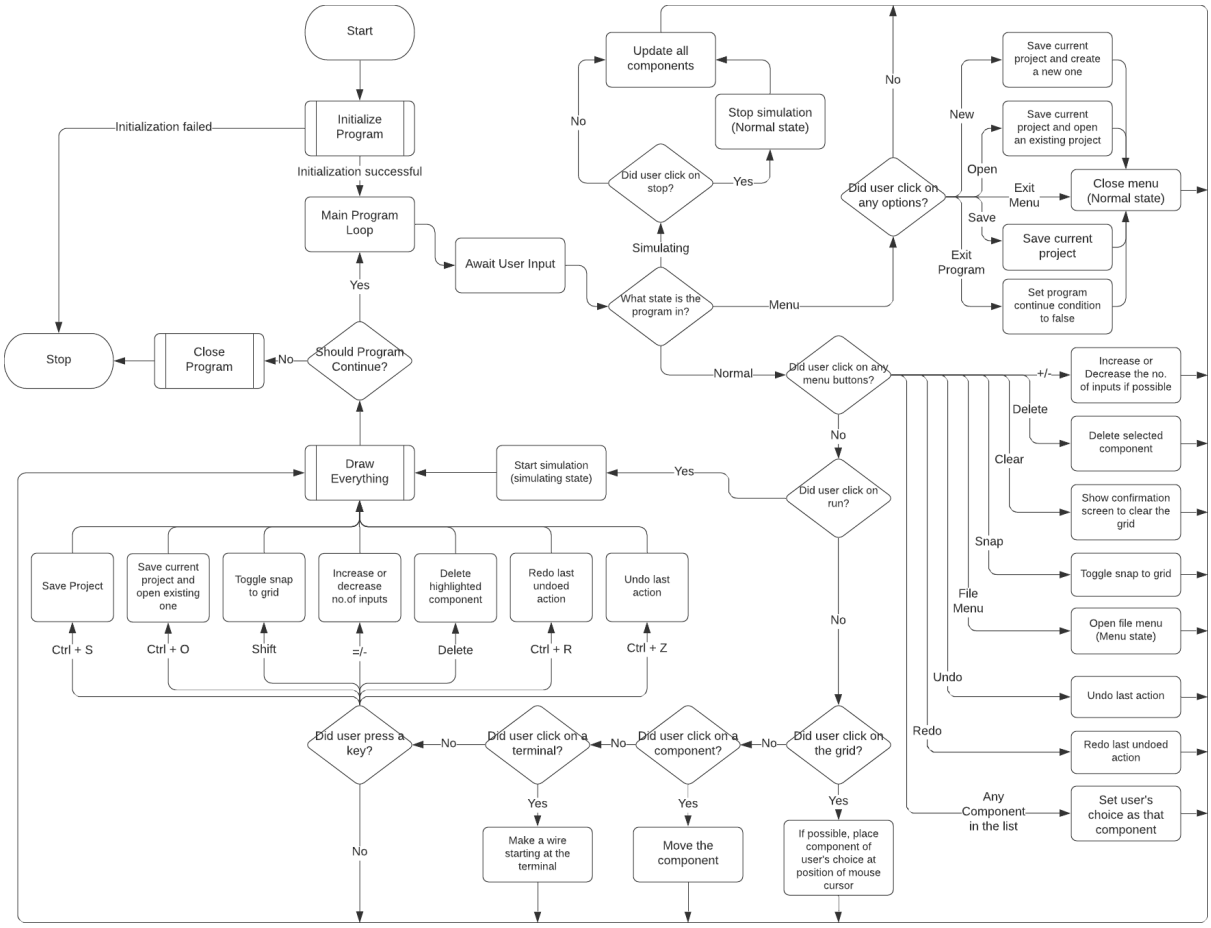

Figure 2.1: Flowchart of the program

2.2 Working of the program

Working of the program can be described in the following three steps:

2.2.1 Initialize

When the program is launched, initialization is carried out. If initialization is unsucessful, the program will display and error and quit. During this process, the libraries used by the program (SDL2 and SDL2_ttf) are initialized. Then the SDL window and renderer are created. Fonts used to display text in the program are loaded and then a character map is created using the fonts. Since SDL2_ttf ant the fonts are no longer required after the character map has been created, the fonts get closed and SDL2_ttf is quit.

Algorithm for Inititalizing

```
initialize SDL
did SDL initialize successfully?
  yes: continue
  no : display error
     exit program
create SDL window
create SDL renderer
were the window and renderer created successfully?
  yes: continue
  no: display error
    exit program
initialize the grid
  fill grid with -1
had user clicked on a file to open?
  read from file and update grid
  update window title
initialize menu
  set dimensions and positions for all buttons
change directory to directory of executable
initialize SDL_ttf
did SDL_ttf initialize successfully?
  yes: continue
  no : display error
       exit program
load fonts
were fonts loaded successfully?
```

```
  yes: continue
  no : display error
       exit program
create character map
  create textures for all characters to be used
close fonts
close SDL_ttf
```

2.2.2 Loop

All user interaction, simulation and rendering happen inside the main program loop. During each iteration of the loop, the program waits for user input and then processes it as shown in the flowchart. Then all components of the program are rendered (drawn) onto the screen. If the simulation is running then all the components get updated. The loop continues to run until the user quits (presses the X button in title bar or presses Alt+F4) or exits program from the menu.

Algorithm for Loop

```
loop if program continue condition is true{
  await user input
  what state is program in?
    normal:
      if user clicked on RUN
        start simulation
      if user clicked on + or user pressed = key
        increase no. of inputs of current choice if possible
      if user clicked on - or user pressed - key
        decrease no. of inputs of current choice if possible
      if user clicked on Undo or user pressed Ctrl + z
        undo last action
      if user clicked on Redo or user pressed Ctrl + r
        redo last undoed action
      if user clicked on Delete Component or user pressed Delete key
        delete selected component on the grid
      if user clicked on Clear Grid
        show confirmation screen
          user clicked yes: empty the grid
      if user clicked on Snap to Grid or user held Shift
        toggle snapping to grid
```

```
      is snapping toggled on?
        yes: set snap button text to "Snap to Grid: On"
        no : set snap button text to "Snap to Grid: Off"
    if user clicked on File Menu
      open the menu
    if user clicked on any component in the list
      set user's choice to be that component
    if user pressed Ctrl + o
      save current project and open existing one
    if user pressed Ctrl + s
      save project
    if user clicked on component on the grid
      move the component
    if user clicked on terminal
      make wire starting at that terminal
    if user clicked on the grid
      if possible place user's choice of component on the grid
      at current mouse cursor position
  simulating:
    if user clicked on STOP
      stop simulation
    update all components
  menu:
    if user clicked on Save:
      save current project
      close menu
    if user clicked on New:
      save current project and create a new one
      close menu
    if user clicked on Open:
      save current project and open an existing one
      close menu
    if user clicked on Exit Menu:
      close menu
    if user clicked on Exit Program:
      set program continue condition to false
      close menu
draw everything
  draw menu
```

```
    draw grid
    draw components
    draw wires
}
```

2.2.3 Close

After the program exits from the loop, it calls some clean up functions which destroy the textures, window and renderer, and free memory.

Algorithm for Closing

```
destroy all textures
destroy window
destroy renderer
quit SDL
exit program
```

2.3 Brief Descriptions of the source files

2.3.1 component.c and component.h

As the name suggests, these files contain all the necessary information about the components used in the program. The header file defines a structure named Component that encompasses the details about a component including its size, position, input source, number of inputs, input state(s), output state(s) and other information which is later used. The output of any component (except clock and state) depends on its input(s). To get the desired output for any component from its inputs, the source file defines different component specific functions. The working of these functions is pretty straight-forward as they follow the standard logic operations available in C. As for the clock, its output is generated based on the value of time variable, which changes as the program progresses, defined in program.c. The clock inverts its current state when time reaches a certain value. The output of state is inverted when the user clicks on it.

2.3.2 draw.h and draw.c

These two files contain the variables and functions that are responsible for drawing all the elements that are visible on the screen such as Buttons, Components, and Wires. It also handles rendering text in the SDL window where necessary. The header file defines

an enumeration of confirmation flags that are later used to ask the user for confirmation on certain operations. The standard rendering functions available in the SDL library are used in order to draw Buttons and Components. However, SDL does not offer the functionality to draw curves. So, a simple algorithm that approximates a cubic Bezier Curve is used to draw wires. As for displaying text, a character map consisting of all the ASCII characters is predefined when the program starts. The font used is robotoo.ttf. The character map is later used to display any text (ASCII based) on the screen.

2.3.3 interaction.h and interaction.c

User interaction is an integral part of any program, even more so for programs that use both mouse and keyboard to take input. These two files are responsible for handling such interactions. The header file defines various structures that are necessary for the Undo/Redo functionalities. The source file defines different functions that determine what will happen when a certain button is pressed or when a component is placed on the grid. Since these functions handle interaction with the user, they are usually only called when an event occurs. An SDL event encompasses mouse clicks, keyboard presses, etc. Different functions are called for different events. This coordination is handled in the file program.c.

2.3.4 program.h and program.c

To keep the main.c file clean, the main program loop is defined in this file. For this reason, it acts as the centerpiece of the program that coordinates the functions of all other files. To begin with, the header file defines macros for configuring the main window and different elements inside it. Also, the colors that are frequently used in the program are defined here. The source file can be vaguely divided into two parts: Initialization and Main Program Loop. The initialization part is responsible for setting up all the necessary elements needed for the program to function properly. This is a one-time process that occurs when the program is launched. The Main Program Loop, as the name suggests, is a loop that runs over and over until the user exits the program. Everything that the user does inside the program is handled in this section. During each loop, the program checks for events, performs necessary operations based on them, updates the elements on the screen if required, and redraws all of those elements.

2.3.5 main.c

As mentioned earlier, this file is kept as clean as possible by defining the main loop in program.c. The main function calls functions for initialization, the main program loop, and finally closing the program.

Chapter 3

IMPLEMENTATION

3.1 The main() function

The main function which can be found in the main.c file only calls the 3 important functions in the program i.e. `InitEverything()`, `MainProgramLoop()` and `CloseEverything`.

```
#include "program.h"

int main(int argc, char **argv)
{
    int grid[GRID_ROW * GRID_COL];
    //Initialize the window, char maps, grid, font, UI etc
    InitEverything(grid, argc, argv);
    //The main program loop. User input, drawing, output, simulation everything
        happens here
    MainProgramLoop(grid);
    //Destroying textures, window, renderer and closing fonts happens here
    CloseEverything();
    return 0;
}
```

3.2 Initializing

Initializing consists of initializing the libraries, the grid, font and the character maps. Initializing is done through the `InitEverything()` function which calls other helper functions to prepare everything as shown below:

```
TTF_Font *font = NULL;         //Font used in UI
TTF_Font *displayFont = NULL; //Font used in decoders
SDL_Texture *characters[127 - 32]; //Character map for UI
```

```
int characterWidth[127 - 32];
SDL_Texture *displayChars[16]; //Character Map for decoders

void InitEverything(int grid[GRID_ROW * GRID_COL], int argc, char **argv)
{
    // Initialize SDL
    if (SDL_Init(SDL_INIT_EVERYTHING) < 0)
        exit(-1);
    // Create Window and renderer
    window = SDL_CreateWindow("MinimaLogic", 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT,
        SDL_WINDOW_MAXIMIZED | SDL_WINDOW_RESIZABLE);
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_SOFTWARE);
    if (!(window && renderer))
        exit(-2);
    // Set minimum size for the window
    SDL_SetWindowMinimumSize(window, MIN_WINDOW_WIDTH, MIN_WINDOW_HEIGHT);
    // Initialize the grid
    InitGrid(grid);
    // If user clicked on a .mlg file and opened it with
    // Minimalogic.exe then read from that file and then update grid and window
        title
    if (argc > 1){
        ReadFromFile(grid, argv[1]);
        fileExists = true;
        SDL_strlcpy(currentFile, argv[1], 256);
        UpdateWindowTitle(argv[1]);
    }
    // Get width and height of the window
    int w, h;
    SDL_GetWindowSize(window, &w, &h);
    // Initialize the Menu
    InitMenu(w, h, false);
    //Change directory to location of the executable so that the program can
        find fonts
    char *path = argv[0], i;
    for (i = SDL_strlen(path) - 1; path[i] != '\\'; i--);
    path[i] = '\0';
    _chdir(path);
    // Initialize fonts
    InitFont();
    // Create character maps
```

```
    CharacterMap();
    // Close fonts and SDL_ttf since they are no longer needed
    TTF_CloseFont(font);
    TTF_CloseFont(displayFont);
    TTF_Quit();
}
```

3.2.1 Initializing Menu

To initialize the menu, the `InitMenu()` function is called which sets positions and dimensions for all buttons in the menu. For the buttons, we have defined a Button struct as follows:

```
typedef struct Button
{
    SDL_Rect buttonRect; // contains the position and dimension of the buttons
    Selection selection;
    SDL_Color color;   // holds the color of the button
} Button;
```

The `SDL_Rect` and `SDL_Color` are structs defined by SDL2. They have been discussed in the Rendering section. We have Selection the selection struct as shown below. It holds the information needed to place a component on the grid.

```
typedef struct
{
    Type type; //Type enum to store type of component
    char size; //size of the component
    Pair pos; //position of the component
} Selection;
```

The Pair struct only stores a pair of integers.

```
typedef struct
{
    int x, y;
} Pair;
```

The `InitMenu()` function is defined as follows:

```
// Declare all buttons used in program as globals
Button confirmYes = {.color = {GREEN, 255}};
Button confirmNo = {.color = {RED, 255}};
Button confirmCancel = {.color = {BLUE, 255}};
```

```
Button Components[g_total];
Button SideMenu[sm_total];
Button FileMenu[fm_total];

void InitMenu(int windowWidth, int windowHeight, bool simulating)
{
    // Set color, position and dimension for all buttons in the side menu
    for (int i = 0; i < sm_total; i++)
    {
        SideMenu[i].buttonRect.w = MENU_WIDTH - 20;
        SideMenu[i].color = (SDL_Color){BLACK};
        SideMenu[i].buttonRect.h = MENU_FONT_SIZE;
        SideMenu[i].buttonRect.x = 10;
        SideMenu[i].buttonRect.y = windowHeight - (sm_total - i) * (10 +
            SideMenu[i].buttonRect.h);
    }
    // Set color, position and dimension for all buttons in the file menu
    for (int i = 0; i < fm_total; i++)
    {
        FileMenu[i].buttonRect.w = MENU_WIDTH - 20;
        FileMenu[i].buttonRect.h = MENU_FONT_SIZE;
        FileMenu[i].buttonRect.x = windowWidth / 2 - FileMenu[i].buttonRect.w /
             2;
        FileMenu[i].buttonRect.y = windowHeight / 2 +
                                   FileMenu[i].buttonRect.h / 2 +
                                   (i - fm_total / 2) * (FileMenu[i].buttonRect.h +
                                        10);
    }
    // Set color of the RUN/STOP button
    // also set its position to be at the top of the screen
    if (simulating)
        SideMenu[sm_run].color = (SDL_Color){RED};
    else
        SideMenu[sm_run].color = (SDL_Color){GREEN};
    SideMenu[sm_run].buttonRect.y = 10;
    // Set Components (dropdown) button to be right below RUN/STOP button
    SideMenu[sm_compo].buttonRect.y = SideMenu[sm_run].buttonRect.y + SideMenu[
        sm_compo].buttonRect.h + 10;

    // Set color, dimension and position of the - (decrease inputs) button
    SideMenu[sm_dec].color = (SDL_Color){RED};
```

```
42      SideMenu[sm_dec].buttonRect.w = 0.15 * MENU_WIDTH - 10;
43      SideMenu[sm_dec].buttonRect.x = 10;
44
45      // Set position and dimension for rect which shows no. of inputs of current
            choice
46      InputsCount.x = SideMenu[sm_dec].buttonRect.x + SideMenu[sm_dec].buttonRect
            .w + 5;
47      InputsCount.y = SideMenu[sm_dec].buttonRect.y;
48      InputsCount.w = 0.7 * MENU_WIDTH - 10;
49      InputsCount.h = MENU_FONT_SIZE;
50
51      // Set color, dimension and position of the + (increase inputs) button
52      SideMenu[sm_inc].color = (SDL_Color){RED};
53      SideMenu[sm_inc].buttonRect.w = 0.15 * MENU_WIDTH - 10;
54      SideMenu[sm_inc].buttonRect.x = InputsCount.x + InputsCount.w + 5;
55      SideMenu[sm_inc].buttonRect.y = SideMenu[sm_dec].buttonRect.y;
56
57      /* Buttons in the confirmation screen */
58      confirmYes.buttonRect.w = 150;
59      confirmYes.buttonRect.h = MENU_FONT_SIZE;
60      confirmYes.buttonRect.x = windowWidth / 2 - 200 + 25;
61      confirmYes.buttonRect.y = windowHeight / 2 - 100 + 200 - confirmYes.
            buttonRect.h - 15 - MENU_FONT_SIZE;
62
63      confirmNo.buttonRect.w = 150;
64      confirmNo.buttonRect.h = MENU_FONT_SIZE;
65      confirmNo.buttonRect.x = windowWidth / 2 - 200 + 400 - 25 - confirmNo.
            buttonRect.w;
66      confirmNo.buttonRect.y = windowHeight / 2 - 100 + 200 - confirmNo.
            buttonRect.h - 15 - MENU_FONT_SIZE;
67
68      confirmCancel.buttonRect.w = 150;
69      confirmCancel.buttonRect.h = MENU_FONT_SIZE;
70      confirmCancel.buttonRect.x = windowWidth / 2 - confirmCancel.buttonRect.w /
            2;
71      confirmCancel.buttonRect.y = windowHeight / 2 - 100 + 200 - confirmCancel.
            buttonRect.h - 10;
72
73      // Buttons in the dropdown
74      for (int i = 0; i < g_total; i++)
75      {
```

```
        Components[i].selection.type = i;
        Components[i].selection.size = 2;
        Components[i].buttonRect.x = 20;
        Components[i].buttonRect.y = SideMenu[sm_compo].buttonRect.y +
                                     SideMenu[sm_compo].buttonRect.h +
                                     i * (CELL_SIZE * SCALE + 2) + 10;
        Components[i].buttonRect.w = MENU_WIDTH - 40;
        Components[i].buttonRect.h = MENU_FONT_SIZE - 10;
    }
}
```

In the above function, all values starting with `sm_` and `fm_` are part of following enums:

```
// enum to keep track of buttons in the side menu
typedef enum {sm_run, sm_compo, sm_inc, sm_dec, sm_undo, sm_redo, sm_snap,
    sm_delete, sm_clear, sm_fmenu, sm_total } SidemenuButtons;
// enum to keep track of buttons in the file menu
typedef enum{ fm_new, fm_open, fm_save, fm_saveas, fm_exitm, fm_exitp,
    fm_total } FileMenuButtons;
```

These enums are also used later to check which button the user clicked.

3.2.2 Initializing the grid

Initializing the grid is a simple process, it only consists of filling the grid with -1 which represents empty space.

```
void InitGrid(int grid[GRID_ROW * GRID_COL])
{
    for (int i = 0; i < GRID_COL * GRID_ROW; i++)
        grid[i] = -1;
}
```

In case the user has opened a .mlg file using Minimalogic, the file will be read and the grid will be updated accordingly by the `ReadFromFile()` function which has been described later in User Interaction section.

3.2.3 Initializing fonts and Creating Character maps

In our program, we are rendering text by displaying a texture containing a character onto the screen. The process of creating a texture for a character and then displaying it to the screen is very heavy. So doing this every time we have to render text will be extermely inefficient. Thus to counter this, we create character maps once at the start of

the program which can be used to render text later without having to create a texture everytime.

```
// function to initialize SDL_ttf and load fonts to be used
static void InitFont()
{
    TTF_Init(); // initialize sdl_ttf
    // load fonts
    font = TTF_OpenFont("ui_font.ttf", 25);
    displayFont = TTF_OpenFont("led_font.otf", 100);
    // exit program if loading fonts fails
    if (font == NULL || displayFont == NULL)
    {
        SDL_Log("Failed to load the font: %s\n", TTF_GetError());
        exit(-3);
    }
}
// function to create character maps
void CharacterMap()
{
    // list of all characters for decoders
    char *nums[16] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B
        ", "C", "D", "E", "F"};
    // surface to be used to create texture
    SDL_Surface *characterSurface;
    SDL_Color white = {WHITE, 200};
    SDL_Color black = {BLACK, 255};

    // create texture for characters having unicode (32 - 127)
    // this includes all characters of english alphabet as well as special
        symbols and numbers
    for (int i = 32; i < 127; i++)
    {
        // render character on surface
        characterSurface = TTF_RenderText_Blended(font, (char *)&i, white);
        // make texture from surface
        characters[i - 32] = SDL_CreateTextureFromSurface(renderer,
            characterSurface);
        // get width of the character (used to retain proper shape of character
            )
        characterWidth[i - 32] = characterSurface ? characterSurface->w : 0;
    }
```

```
36      // similar process for characters to be shown in decoders
37      for (int i = 0; i < 16; i++)
38      {
39          characterSurface = TTF_RenderText_Blended(displayFont, nums[i], black);
40          displayChars[i] = SDL_CreateTextureFromSurface(renderer,
                characterSurface);
41      }
42      // free the surface to free memory
43      SDL_FreeSurface(characterSurface);
44  }
```

3.3 Main Program Loop

Everything the user sees and does happens in the main program loop. In our program this loop has been defined inside the `MainProgramLoop()` function. The outline of the loop has been given below: The particular steps in the loop have been explained in the following sections.

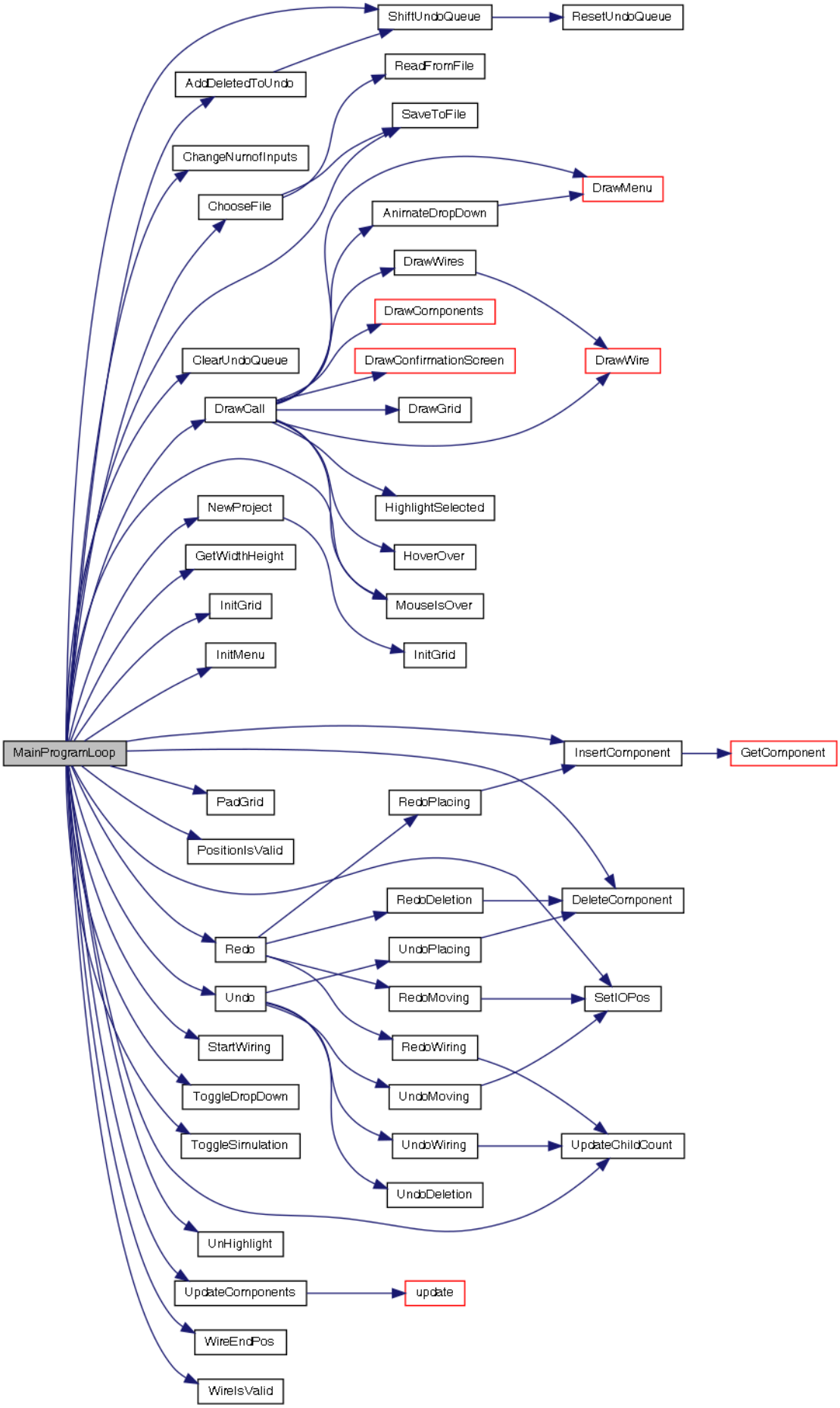

Figure 3.1: Call tree for MainProgramLoop

Code for the main program loop:

```
Actions UndoBuffer[MAX_UNDOS];
void MainProgramLoop(int grid[GRID_ROW * GRID_COL])
{
    // defining various fields required during the loop
    Selection compoChoice = {.type = 0, .size = 0};
    Pair selected = {-1, -1};

    int x, y;
    Pair gridPos;
    int pad_x, pad_y;
    PadGrid(&pad_x, &pad_y);

    int sender, receiver, sendIndex, receiveIndex, compoMoved;
    bool simulating = false, menuExpanded = false, drawingWire = false,
        movingCompo = false, confirmWire = false;
    bool snapToGrid = false, snapToggeled = false, cursorInGrid, draw, updated
        = false, ctrlHeld = false;
    char dropDownAnimationFlag = 0, startAt = 0, endAt = 0, animating = 8;
    Pair offset, initialPos;
    ConfirmationFlags confirmationScreenFlag = none;
    unsigned char updateOrder[256];

    int currentUndoLevel = 0, totalUndoLevel = 0;
    bool run = true;

    SDL_Event e;
    while (run)
    {
        // to make sure the program runs at consistent fps
        int begin = SDL_GetTicks();
        // to limit how many times drawing is done
        draw = !simulating;
        // get mouse cursor position
        SDL_GetMouseState(&x, &y);
        // get position of mouse relative to the grid
        if (x - pad_x > 0)
            gridPos.x = (x - pad_x) / CELL_SIZE;
        else
            gridPos.x = -1;
        if (y - pad_y > 0)
```

```
39          gridPos.y = (y - pad_y) / CELL_SIZE;
40      else
41          gridPos.y = -1;
42      // mod the position of mouse by SCALE/2 if snapping is toggled
43      // this will make the component preview snap to half of the gridlines
44      if (snapToGrid && gridPos.x >= 0 && gridPos.y >= 0)
45      {
46          gridPos.x -= gridPos.x % (SCALE / 2);
47          gridPos.y -= gridPos.y % (SCALE / 2);
48      }
49      // check wheter cursor is in the grid or not
50      cursorInGrid = gridPos.x >= 0 && gridPos.x < GRID_ROW && gridPos.y >= 0
            &&
51                  gridPos.y < GRID_COL;
52      // getting and handling user input
53      while (SDL_WaitEventTimeout(&e, DELAY / 10))
54      {
55          /* see User Interaction section */
56          // Draw once and only if any updates occur
57          if (draw)
58          {
59              DrawCall(menuExpanded, drawingWire, x, y, compoChoice, pad_x,
                    pad_y,
60                       simulating, &dropDownAnimationFlag, gridPos, grid,
                           movingCompo, selected, snapToGrid,
                           confirmationScreenFlag);
61              draw = false;
62          }
63      }
64      // updating components if simulation is running
65      if (simulating)
66      {
67          for (int i = 0; i < 256; i++)
68              AlreadyUpdated[i] = false;
69          UpdateComponents(updateOrder);
70          time += DELAY;
71          if (time >= DELAY * 20)
72              time = 0;
73          selected = (Pair){-1, -1};
74      }
75      // Always draw if simulation is running or an animation is running
```

```
76          if (simulating || animating < 8)
77              DrawCall(menuExpanded, drawingWire, x, y, compoChoice, pad_x, pad_y,
78                      simulating, &dropDownAnimationFlag, gridPos, grid,
                            movingCompo, selected, snapToGrid, confirmationScreenFlag
                            );
79          // to keep track of the animation
80          animating += 1;
81          animating = (animating > 8)? 8 : animating;
82          // Delay the loop to limit frame rate and redude load on cpu
83          if ((SDL_GetTicks() - begin) < DELAY)
84              SDL_Delay(DELAY - (SDL_GetTicks() - begin));
85          else
86              SDL_Delay(DELAY);
87      }
88  }
```

In order to optimize our program and make it more efficient, we have tried to render/draw things as few times as possible. This has been obtained by rendering only if user provides some imput or the simulation or animation is running. Doing this dramatically lessens the load on the CPU.

3.3.1 The Components

To make the components we have defined a Component (_component) struct as follows:

```
1   typedef struct _component
2   {
3       Pair start,      // position of the component on the grid
4            inpPos[MAX_INPUTS], // position of input terminals
5            outPos[MAX_TERM_NUM], // position of output terminals
6            inpSrc[MAX_INPUTS]; // source for inputs array also used to make wires
7       unsigned char size,  // height of the component
8                     width,   // width of the component
9                     inum,    // no. of inputs
10                    onum,    // no. of outputs
11                    childCount;  // no.of components which directly or indirectly
                          get input from this component
12      bool outputs[MAX_TERM_NUM]; // to keep track of outputs after updating
13      Type type;    //Type of component
14      struct _component *inputs[MAX_INPUTS]; //pointer to Components providing
            input
15  } Component;
```

By using an array of pointers to keep track of inputs, we have applied a dort of backwards linked list approach which allows us to prevent errors while updating the components. The type of component is determined by the following enum:

```
typedef enum{ state, probe, clock, g_not, d_oct, d_4x16, g_and, g_or, g_nand,
    g_nor, g_xor, g_xnor, g_total} Type;
```

Updating A Component

Updating a component is implemented in the following way:

```
// declaring function for each component
static void Tick(Component *component);
static void orGate(Component *component);
static void andGate(Component *component);
static void notGate(Component *component);
static void norGate(Component *component);
static void xorGate(Component *component);
static void xnorGate(Component *component);
static void nandGate(Component *component);
static void DoNothing(Component *component);
static void Decode(Component *component);
static int BinToDec(bool[4]);
// to keep track of components being updated and prevent infinite recursion
bool AlreadyUpdated[256];

// array of function pointers so that we can call appropriate function
    according to the type of component to update it
static void (*operate[g_total])(Component *component) = {DoNothing, DoNothing,
     Tick, notGate, Decode, Decode, andGate, orGate, nandGate, norGate, xorGate
    , xnorGate};

// function to update the components providing input
// by using the backwards linked list approach we first update the components
    providing input so that correct output can be produced
static void SetInputs(Component *component)
{
    for (int i = 0; i < component->inum; i++)
    {
        if (component->inpSrc[i].x != -1)
        {
            // to update component only once and prevent stack overflow via
                recursion
            if (!AlreadyUpdated[component->inpSrc[i].x])
```

```
            {
                AlreadyUpdated[component->inpSrc[i].x] = true;
                update(component->inputs[i]);
            }
        }
    }
}
// This function first calls the SetInputs() function and then calls
    appropriate function according to type of array
void update(Component *component)
{
    SetInputs(component);
    operate[component->type](component);
}

// function for and gate
static void andGate(Component *component)
{
    // set initial value if possible else set false as default value
    if (component->inpSrc[0].x >= 0){
        component->outputs[0] = component->inputs[0]->outputs[component->inpSrc
            [0].y];
    }
    else{
        component->outputs[0] = false;
    }
    // loop through all inputs and AND all of them to get output
    for (int i = 1; i < component->inum; i++){
        if (component->inpSrc[i].x >= 0){
            component->outputs[0] = component->outputs[0] && component->inputs[i
                ]->outputs[component->inpSrc[i].y];
        }
        else{
            component->outputs[0] = false;
            break;
        }
    }
}
// other components work similarly
```

Here, the working of And gate has been described. Other gates are also updated similarly which can be seen in the full source code. **Updating All Component**

As seen above, components placed on the grid only update if the simulation is running. Updating all the components happens through the `UpdateComponents()` function. This function loop through the list of components and updates all of them once using the `update()` function shown above.

```
static void UpdateComponents(unsigned char *updateOrder){
    for (int i = 0; i < componentCount; i++){
        unsigned char index = updateOrder[i];
        if (!AlreadyUpdated[index]){
            AlreadyUpdated[index] = true;
            update(&ComponentList[index]);
        }
    }
}
```

The `updateOrder` seen above is an array specifying the order in which the components should be updated. This is done to prevent jittering. The order is determining by sorting the components according to their `childCount`.

3.4 User Interaction

Getting and handling user interaction is the biggest portion of our program. This is done in a loop nested inside the main program loop. In this loop we wait for user input and then utilize `switch ... case` statements to respond to the inputs.

```
// wait for user input
while (SDL_WaitEventTimeout(&e, DELAY / 10))
{
    switch (e.type)
    {
    // ask user if the want to save if they quit if they have made changes
    // else just set run to false which ends the main program loop
    case SDL_QUIT:
        if (fileExists && updated)
            confirmationScreenFlag = q_saveChanges;
        else if (updated && componentCount > 0)
            confirmationScreenFlag = q_saveNewFile;
        else
            run = false;
    case SDL_WINDOWEVENT:
    {
        // rearrange the grid and menu if window is resized
```

```
18          int w, h;
19          SDL_GetWindowSize(window, &w, &h);
20          InitMenu(w, h, simulating);
21          PadGrid(&pad_x, &pad_y);
22          break;
23      }
24      case SDL_MOUSEBUTTONDOWN:
25          // handling mouse clicks
26          if (!confirmationScreenFlag)
27          {
28              // this code is executed only if no confirmation screen or menu is
                    shown
29              if (e.button.button == SDL_BUTTON_RIGHT)
30              {
31                  selected = (Pair){-1, -1};
32                  break;
33              }
34              if (cursorInGrid)
35              {
36                  if (!WireIsValid(grid, gridPos, x, y, pad_x, pad_y) && cell(
                        gridPos.y, gridPos.x) >= 0)
37                  {
38                      // this code is executed if user clicked on a component
39                      // flip value of state and clock if they are clicked
40                      if (ComponentList[cell(gridPos.y, gridPos.x)].type == state
                            || (ComponentList[cell(gridPos.y, gridPos.x)].type ==
                            clock))
41                          ComponentList[cell(gridPos.y, gridPos.x)].outputs[0] = !
                                ComponentList[cell(gridPos.y, gridPos.x)].outputs[0];
42                      if (!drawingWire && !movingCompo && !simulating)
43                      {
44                          // select component (used for deleting)
45                          selected = gridPos;
46                          // the following code is used to start move a placed
                                component around on a grid
47                          // which component?
48                          Component compo = ComponentList[cell(gridPos.y, gridPos.x)
                                ];
49                          // initial position of the component
50                          initialPos = compo.start;
51                          // offset from cursor to the topleft corner of component
```

```
52                      offset = (Pair){gridPos.x - initialPos.x, gridPos.y -
                            initialPos.y};
53                      // which component?
54                      compoMoved = cell(gridPos.y, gridPos.x);
55                      // signify that we are moving a component now
56                      movingCompo = true;
57                      // empty the space being occupied by the component
58                      for (int i = initialPos.y; i < initialPos.y + compo.size;
                            i++)
59                          for (int j = initialPos.x; j < initialPos.x + compo.
                                width; j++)
60                              cell(i, j) = -1;
61                  }
62              }
63              else
64              {
65                  selected = (Pair){-1, -1};
66              }
67              // dimension of component user is trying to place
68              int w, h;
69              GetWidthHeight(&w, &h, compoChoice.type, compoChoice.size);
70              // Check if component should be placed
71              if (componentCount < 255 && !simulating && !drawingWire &&
                    PositionIsValid(grid, w, h, compoChoice.pos) && !movingCompo)
72              {
73                  // place the component
74                  InsertComponent(grid, compoChoice, w, h);
75                  // signify that an update has been made
76                  updated = true;
77                  // add this action to undo buffer so that it can be undoed
78                  ShiftUndoBuffer(&currentUndoLevel, &totalUndoLevel);
79                  UndoBuffer[0].act = 'p';
80                  UndoBuffer[0].Action.placed.component = ComponentList[
                        componentCount - 1];
81              }
82              else if (!simulating && !drawingWire && !movingCompo)
83              {
84                  // If user clicks on a terminal start drawing a wire
85                  startAt = WireIsValid(grid, gridPos, x, y, pad_x, pad_y);
86                  if (startAt < 0)
87                  {
```

```
                    // wire begins at a output terminal
                    sender = cell(gridPos.y, gridPos.x);
                    sendIndex = startAt;
                    drawingWire = StartWiring((Pair){x, y});
                }
                else if (startAt > 0)
                {
                    // wire begins at input terminal
                    receiver = cell(gridPos.y, gridPos.x);
                    receiveIndex = startAt;
                    drawingWire = StartWiring((Pair){x, y});
                }
            }
        }
        // cursor is in the menu
        if (x <= MENU_WIDTH)
        {
            // which button was clicked
            Pair clickedButton = MouseIsOver(x, y, menuExpanded, compoChoice,
                 confirmationScreenFlag == fileMenuFlag);
            if (clickedButton.x == sm)
            {
                // button in side menu was clicked
                if (clickedButton.y == sm_run)
                {
                    // toggle simulation if RUN/STOP button is clicked
                    ToggleSimulation(&simulating, updateOrder);
                    selected = (Pair){-1, -1};
                }
                else if (!simulating)
                {
                    // these buttons cant be clicked if simulation is running
                    switch (clickedButton.y)
                    {
                    case (sm_clear):
                        // clear button was clicked
                        // show confirmation screen
                        confirmationScreenFlag = clearGrid;
                        break;
                    case (sm_compo):
                        // component button was clicked
```

```
128                         // toggle dropdown list
129                         ToggleDropDown(&menuExpanded, &dropDownAnimationFlag);
130                         animating = 0;
131                         break;
132                     case (sm_dec):
133                         // decrease input if possible
134                         ChangeNumofInputs(true, &compoChoice);
135                         break;
136                     case (sm_inc):
137                         // increase input if possible
138                         ChangeNumofInputs(false, &compoChoice);
139                         break;
140                     case (sm_delete):
141                         // delete button clicked
142                         // delete the selected component
143                         // add the action to undo buffer
144                         AddDeletedToUndo(&currentUndoLevel, &totalUndoLevel,
                                cell(selected.y, selected.x));
145                         DeleteComponent(grid, selected);
146                         selected = (Pair){-1, -1};
147                         updated = true;
148                         break;
149                     case (sm_undo):
150                         // undo button clicked
151                         // undo if possible
152                         if (currentUndoLevel < totalUndoLevel)
153                             Undo(grid, &currentUndoLevel, totalUndoLevel);
154                         break;
155                     case (sm_redo):
156                         // redo button clicked
157                         // redo if possible
158                         if (currentUndoLevel > 0)
159                             Redo(grid, &currentUndoLevel, totalUndoLevel);
160                         break;
161                     case (sm_snap):
162                         // snap button clicked
163                         // toggle snapping to grid
164                         snapToGrid = !snapToGrid;
165                         snapToggeled = !snapToggeled;
166                         break;
167                     case (sm_fmenu):
```

```
                        // open the file menu
                        confirmationScreenFlag = fileMenuFlag;
                        break;
                    default:
                        break;
                    }
                }
            }
            else if (clickedButton.x == cm && menuExpanded)
            {
                // one of the components in the dropdown component list was
                    clicked
                // update users choice of component to be place
                UnHighlight(compoChoice.type);
                compoChoice = Components[clickedButton.y].selection;
            }
        }
    }
    else
    {
        // a confirmation box or menu is shown
        Pair clickedButton = MouseIsOver(x, y, menuExpanded, compoChoice,
            confirmationScreenFlag == fileMenuFlag);
        char fname[256]; //name of file
        if (confirmationScreenFlag == fileMenuFlag && clickedButton.x == fm)
        {
            // file menu is open
            switch (clickedButton.y)
            {
            case fm_new:
                // new button clicked
                // ask to save if changes have been made
                // then clear the grid for new project
                if (fileExists && updated)
                    confirmationScreenFlag = n_saveChanges;
                else if (updated && componentCount > 0)
                    confirmationScreenFlag = n_saveNewFile;
                else
                {
                    NewProject(grid, &updated);
                    confirmationScreenFlag = none;
```

```
207                 }
208                 break;
209             case fm_open:
210                 // open button clicked
211                 // ask to save if changes have been made
212                 // then open existing file
213                 SDL_strlcpy(fname, currentFile, 256);
214                 if (fileExists && updated)
215                     confirmationScreenFlag = o_saveChanges;
216                 else if (updated && componentCount > 0)
217                     confirmationScreenFlag = o_saveNewFile;
218                 else
219                 {
220                     ChooseFile(grid, false);
221                     confirmationScreenFlag = none;
222                 }
223                 break;
224             case fm_save:
225                 // save button clicked
226                 // save to file if it exists
227                 // else make and save to new file
228                 if (fileExists)
229                     SaveToFile(grid, currentFile);
230                 else
231                     ChooseFile(grid, true);
232                 updated = false;
233                 confirmationScreenFlag = none;
234                 break;
235             case fm_saveas:
236                 // save to a new file
237                 ChooseFile(grid, true);
238                 updated = false;
239                 confirmationScreenFlag = none;
240                 break;
241             case fm_exitm:
242                 // close the menu
243                 confirmationScreenFlag = none;
244                 break;
245             case fm_exitp:
246                 // exit program button clicked
247                 // ask to save if changes have been made
```

```
248                 // then set run to false
249                 if (fileExists && updated)
250                     confirmationScreenFlag = q_saveChanges;
251                 else if (updated && componentCount > 0)
252                     confirmationScreenFlag = q_saveNewFile;
253                 else
254                     run = false;
255                 break;
256             default:
257                 break;
258             }
259         }
260         else if (clickedButton.x == con && clickedButton.y > 0)
261         {
262             // confirmation screen and user clicked on yes
263             switch (confirmationScreenFlag)
264             {
265                 // confirmation screen for what?
266             case clearGrid:
267                 // empty the grid
268                 componentCount = 0;
269                 InitGrid(grid);
270                 updated = true;
271                 break;
272             case q_saveChanges:
273                 // save changes on quitting
274                 SaveToFile(grid, currentFile);
275                 run = false;
276                 break;
277             case q_saveNewFile:
278                 // save changes to new file on quitting
279                 ChooseFile(grid, true);
280                 run = false;
281                 break;
282             case o_saveChanges:
283                 // save changes on opening existing file
284                 SaveToFile(grid, currentFile);
285                 ChooseFile(grid, false);
286                 if (SDL_strcmp(fname, currentFile))
287                 {
288                     ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
```

```
                    updated = false;
                }
                break;
            case o_saveNewFile:
                // save changes to new file on opening existing file
                ChooseFile(grid, true);
                ChooseFile(grid, false);
                if (SDL_strcmp(fname, currentFile))
                {
                    ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
                    updated = false;
                }
                break;
            case n_saveChanges:
                // save changes on new file
                SaveToFile(grid, currentFile);
                NewProject(grid, &updated);
                ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
                updated = false;
                break;
            case n_saveNewFile:
                // save changes to new file on new file
                ChooseFile(grid, true);
                NewProject(grid, &updated);
                ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
                updated = false;
                break;
            default:
                break;
            }
            //close confirmation screen
            confirmationScreenFlag = none;
        }
        else if (clickedButton.x == con && !clickedButton.y)
        {
            // confirmation screen and user clicked on no
            // similar to when yes is clicked but no changes will be saved
            if (confirmationScreenFlag == q_saveChanges ||
                confirmationScreenFlag == q_saveNewFile)
                run = false;
            else if (confirmationScreenFlag == o_saveChanges ||
```

```
                confirmationScreenFlag == o_saveNewFile)
329             {
330                 ChooseFile(grid, false);
331                 if (SDL_strcmp(fname, currentFile))
332                 {
333                     updated = false;
334                     ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
335                 }
336             }
337             else if (confirmationScreenFlag == n_saveChanges ||
                    confirmationScreenFlag == n_saveNewFile)
338             {
339                 NewProject(grid, &updated);
340                 updated = false;
341                 ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
342             }
343             confirmationScreenFlag = none;
344         }
345         else if (clickedButton.x == con)
346             confirmationScreenFlag = none;
347     }
348     break;
349 case SDL_MOUSEBUTTONUP:
350     // user stopped pressing mouse button
351     if (drawingWire)
352     {
353         //if user was drawing wire
354         // check if wire can end where user left holding the mouse button
355         endAt = WireIsValid(grid, gridPos, x, y, pad_x, pad_y);
356         if (endAt && startAt != endAt)
357         {
358             // starting and ending component should not be same
359             if (endAt < 0 && startAt > 0)
360             {
361                 // wire starts at input and ends at output terminal
362                 sender = cell(gridPos.y, gridPos.x);
363                 sendIndex = endAt;
364                 confirmWire = true;
365             }
366             else if (endAt > 0 && startAt < 0)
367             {
```

```
                // wire starts at output and ends at input terminal
                receiver = cell(gridPos.y, gridPos.x);
                receiveIndex = endAt;
                confirmWire = true;
            }
            if (sender != receiver && confirmWire)
            {
                // wire can be made
                // update the childcount for sender
                // update inpSrc and input for receiver
                if (ComponentList[receiver].inpSrc[receiveIndex - 1].x != -1)
                    UpdateChildCount(sender, false);
                ComponentList[receiver].inpSrc[receiveIndex - 1] = (Pair){
                    sender, sendIndex * -1 - 1};
                ComponentList[receiver].inputs[receiveIndex - 1] = &
                    ComponentList[sender];
                updated = true;
                for (int i = 0; i < 256; i++)
                    AlreadyUpdated[i] = false;
                // Add this action to the undo buffer
                UpdateChildCount(sender, true);
                ShiftUndoBuffer(&currentUndoLevel, &totalUndoLevel);
                UndoBuffer[0].act = 'w';
                UndoBuffer[0].Action.wired.sender = sender;
                UndoBuffer[0].Action.wired.connection.receiver = receiver;
                UndoBuffer[0].Action.wired.connection.receiveIndex =
                    receiveIndex - 1;
                UndoBuffer[0].Action.wired.connection.sendIndex = sendIndex *
                     -1 - 1;
            }
        }
        // stop drawing the wire
        drawingWire = false;
    }
    if (movingCompo)
    {
        // if a component was being moved
        Component compo = ComponentList[compoMoved];
        Pair finalPos = initialPos;
        if (compo.start.x < 0 || compo.start.y < 0)
        {
```

```
405             // reset position of component if it goes outside the grid
406             ComponentList[compoMoved].start = initialPos;
407             SetIOPos(&ComponentList[compoMoved]);
408         }
409         if (!PositionIsValid(grid, compo.width, compo.size, compo.start))
410         {
411             // reset position of component if its position is invalid (
                    overlapping)
412             ComponentList[compoMoved].start = initialPos;
413             SetIOPos(&ComponentList[compoMoved]);
414         }
415         else
416             finalPos = compo.start; // give the component a new position if
                    position is valid
417         // update the grid
418         for (int i = finalPos.y; i < finalPos.y + compo.size; i++)
419             for (int j = finalPos.x; j < finalPos.x + compo.width; j++)
420                 cell(i, j) = compoMoved;
421         // no longer moving a component
422         movingCompo = false;
423         selected = finalPos;
424         // add action to undo buffer if fincal and initial components are
                not same
425         if (initialPos.x != compo.start.x || initialPos.y != compo.start.y)
426         {
427             updated = true;
428             ShiftUndoBuffer(&currentUndoLevel, &totalUndoLevel);
429             UndoBuffer[0].act = 'm';
430             UndoBuffer[0].Action.moved.before = initialPos;
431             UndoBuffer[0].Action.moved.after = finalPos;
432             UndoBuffer[0].Action.moved.index = compoMoved;
433         }
434     }
435 case SDL_MOUSEMOTION:
436 {
437     // get width and height of user's choice of component
438     int w, h;
439     GetWidthHeight(&w, &h, compoChoice.type, compoChoice.size);
440     compoChoice.pos = gridPos;
441     if (compoChoice.pos.x + w >= GRID_ROW)
442         compoChoice.pos.x = GRID_ROW - w;
```

```
443         if (compoChoice.pos.y + h >= GRID_COL)
444             compoChoice.pos.y = GRID_COL - h;
445     }
446         if (drawingWire)
447         {
448             // update ending position of wire
449             WireEndPos(x, y);
450         }
451         if (movingCompo)
452         {
453             // update position of component being moved
454             Component compo = ComponentList[compoMoved];
455             Pair newPos = {gridPos.x - offset.x, gridPos.y - offset.y};
456             if (gridPos.x - offset.x + compo.width >= GRID_ROW)
457                 newPos.x = GRID_ROW - compo.width;
458             if (gridPos.y - offset.y + compo.size >= GRID_COL)
459                 newPos.y = GRID_COL - compo.size;
460             if (snapToGrid)
461             {
462                 newPos.x -= newPos.x % (SCALE / 2);
463                 newPos.y -= newPos.y % (SCALE / 2);
464             }
465             newPos.x = newPos.x < 0 ? 0 : newPos.x;
466             newPos.y = newPos.y < 0 ? 0 : newPos.y;
467 
468             compo.start = newPos;
469             CollisionCheck(grid, &compo);
470             ComponentList[compoMoved].start = compo.start;
471             SetIOPos(&ComponentList[compoMoved]);
472         }
473         break;
474     case SDL_KEYDOWN:
475         // handling key pressess
476         // these are just shortcuts for the buttons
477         switch (e.key.keysym.scancode)
478         {
479         case SDL_SCANCODE_MINUS:
480             ChangeNumofInputs(true, &compoChoice);
481             break;
482         case SDL_SCANCODE_EQUALS:
483             ChangeNumofInputs(false, &compoChoice);
```

```
484             break;
485         case SDL_SCANCODE_DELETE:
486             if (!simulating)
487             {
488                 if (cursorInGrid && cell(gridPos.y, gridPos.x) >= 0)
489                 {
490                     AddDeletedToUndo(&currentUndoLevel, &totalUndoLevel, cell(
                            gridPos.y, gridPos.x));
491                     DeleteComponent(grid, gridPos);
492                     updated = true;
493                 }
494                 else if (selected.x >= 0)
495                 {
496                     AddDeletedToUndo(&currentUndoLevel, &totalUndoLevel, cell(
                            selected.y, selected.x));
497                     DeleteComponent(grid, selected);
498                     updated = true;
499                 }
500                 selected = (Pair){-1, -1};
501             }
502             break;
503         case SDL_SCANCODE_LSHIFT:
504             if (!snapToggeled)
505                 snapToGrid = true;
506             break;
507         case SDL_SCANCODE_RSHIFT:
508             if (!snapToggeled)
509                 snapToGrid = true;
510             break;
511         case SDL_SCANCODE_LCTRL:
512             ctrlHeld = !simulating;
513             break;
514         case SDL_SCANCODE_RCTRL:
515             ctrlHeld = !simulating;
516             break;
517         case SDL_SCANCODE_Z:
518             if (ctrlHeld && currentUndoLevel < totalUndoLevel)
519                 Undo(grid, &currentUndoLevel, totalUndoLevel);
520             break;
521         case SDL_SCANCODE_R:
522             if (ctrlHeld && currentUndoLevel > 0)
```

```
523             Redo(grid, &currentUndoLevel, totalUndoLevel);
524         break;
525     case SDL_SCANCODE_S:
526         if (ctrlHeld)
527         {
528             if (fileExists)
529                 SaveToFile(grid, currentFile);
530             else
531                 ChooseFile(grid, true);
532             updated = false;
533         }
534         break;
535     case SDL_SCANCODE_O:
536         if (ctrlHeld && !simulating)
537         {
538             if (fileExists && updated)
539                 confirmationScreenFlag = o_saveChanges;
540             else if (updated && componentCount > 0)
541                 confirmationScreenFlag = o_saveNewFile;
542             else
543                 ChooseFile(grid, false);
544             ClearUndoBuffer(&currentUndoLevel, &totalUndoLevel);
545             updated = false;
546         }
547         break;
548     default:
549         break;
550     }
551     break;
552 case SDL_KEYUP:
553     switch (e.key.keysym.scancode)
554     {
555     case SDL_SCANCODE_LSHIFT:
556         snapToGrid = snapToggeled;
557         break;
558     case SDL_SCANCODE_RSHIFT:
559         snapToGrid = snapToggeled;
560         break;
561     case SDL_SCANCODE_LCTRL:
562         ctrlHeld = false;
563         break;
```

```
564         case SDL_SCANCODE_RCTRL:
565             ctrlHeld = false;
566             break;
567         default:
568             break;
569         }
570         break;
571     default:
572         break;
573     }
574     /* draw */
575 }
```

3.4.1 Placing The Components

Placing is done by following function. It gets an appropriate component according to the users choice and adds the component to the component list and updated the grid.

```
1  // macro to make indexing grid easier
2  #define cell(y, x) grid[y * GRID_ROW + x]
3  //These are helper functions which set the values for membersoof component
       struct as per users choice
4  static Component MultiInputComponent(Type type, int inpNum, Pair pos)
5  {
6      Component component;
7      component.start = pos;
8      component.size = inpNum;
9      component.inum = inpNum;
10     component.onum = 1;
11     component.width = 4;
12     component.size *= SCALE;
13     component.width *= SCALE;
14     component.childCount = 0;
15     component.type = type;
16     ClearIO(&component);
17     SetIOPos(&component);
18     return component;
19 }
20 static Component SingleInputComponent(Type type, Pair pos);
21 static Component MultiOutComponent(Type type, Pair pos);
22
23 // call appropriate helper function to match user choice
```

```
Component GetComponent(Type type, char inpNum, Pair pos)
{
    if (type <= g_not)
        return SingleInputComponent(type, pos);
    else if (type < g_and)
        return MultiOutComponent(type, pos);
    else
        return MultiInputComponent(type, inpNum, pos);
}

// update the list and the grid
void InsertComponent(int *grid, Selection choice, int width, int height)
{
    ComponentList[componentCount] =
        GetComponent(choice.type, choice.size, choice.pos);
    for (int y = choice.pos.y; y < choice.pos.y + height; y++)
    {
        for (int x = choice.pos.x; x < choice.pos.x + width; x++)
        {
            cell(y, x) = componentCount;
        }
    }
    componentCount++;
}
```

3.4.2 Deleting a component

Deleting a component is done by the following function:

```
void DeleteComponent(int *grid, Pair gridPos)
{
    if (cell(gridPos.y, gridPos.x) == -1 || gridPos.x < 0 || gridPos.y < 0)
        return;
    // delete which component?
    int toDelete = cell(gridPos.y, gridPos.x);
    // empty the space occupied by that component in grid
    // decrease index of components whose index was higher than the one being
        deleted
    for (int i = 0; i < GRID_COL; i++)
    {
        for (int j = 0; j < GRID_ROW; j++)
        {
```

```
            if (cell(i, j) == toDelete)
                cell(i, j) = -1;
            else if (cell(i, j) > toDelete)
                cell(i, j)--;
        }
    }
    // if a component was getting input from the deleted component then set its
        imput source to empty
    // update input source of components whose inputs had index higher than the
        one being deleted
    for (int i = 0; i < componentCount; i++)
    {
        Component *compo = &ComponentList[i];
        for (int j = 0; j < compo->inum; j++)
        {
            if (compo->inpSrc[j].x == toDelete)
            {
                compo->inpSrc[j] = (Pair){-1, -1};
                compo->inputs[j] = NULL;
            }
            else if (compo->inpSrc[j].x > toDelete)
            {
                compo->inpSrc[j].x--;
                compo->inputs[j] = &ComponentList[compo->inpSrc[j].x];
            }
        }
    }
    // shift all the components having index higher than the one deleted
        forward in the list
    for (int i = toDelete; i < componentCount - 1; i++)
    {
        ComponentList[i] = ComponentList[i + 1];
    }
    componentCount--;
}
```

3.4.3 Undo & Redo

Undo and Redo both rely on the undo buffer. Every time an action is taken, the changes made by it get pushed to the start of the buffer. This can be seen in the User Interaction code. To keep track of the actions we have defined a seperate struct for each as follows:

```
typedef struct
{
    unsigned char sendIndex, receiver, receiveIndex;
} Connection;

typedef struct
{
    unsigned char index, conNo;
    Component deletedCompo;
    Connection connections[255];
} Delete;

typedef struct
{
    unsigned char sender;
    Connection connection;
} Wiring;
typedef struct
{
    Component component;
} Place;

typedef struct
{
    unsigned char index;
    Pair before, after;
} Move;

typedef struct
{
    char act;
    union
    {
        Delete deleted;
        Wiring wired;
        Place placed;
        Move moved;
    } Action;
} Actions;
```

A union has been defined in the Actions struct so that only one type of action may be

represented and to make it more memory efficient.

Everytime we undo, we first undo the action and then move forwards in the undo buffer i.e. "backwards in time". The `Undo` function undos the changes at current index in the undo buffer and moves forward in the buffer(index increases). A seperate helper function has been defined to undo each type of action. Depending on the type of action appropriate helper function is called.

```
static void UndoDeletion(Delete deleted, int *grid)
{
    int toDelete = deleted.index;

    for (int i = 0; i < GRID_COL; i++)
    {
        for (int j = 0; j < GRID_ROW; j++)
        {
            if (cell(i, j) >= deleted.index)
                cell(i, j)++;
        }
    }
    componentCount++;

    for (int i = 0; i < componentCount; i++)
    {
        Component *compo = &ComponentList[i];
        for (int j = 0; j < compo->inum; j++)
        {
            if (compo->inpSrc[j].x >= toDelete)
            {
                compo->inpSrc[j].x++;
                compo->inputs[j] = &ComponentList[compo->inpSrc[j].x];
            }
        }
    }
    for (int i = componentCount; i > deleted.index; i--)
    {
        ComponentList[i] = ComponentList[i - 1];
    }

    for (int i = 0; i < deleted.conNo; i++)
    {
        ComponentList[deleted.connections[i].receiver].inpSrc[deleted.
            connections[i].receiveIndex] = (Pair){deleted.index, deleted.
```

```
            connections[i].sendIndex};
35          ComponentList[deleted.connections[i].receiver].inputs[deleted.
                connections[i].receiveIndex] = &ComponentList[deleted.index];
36      }
37      ComponentList[deleted.index] = deleted.deletedCompo;
38      for (int i = deleted.deletedCompo.start.x; i < deleted.deletedCompo.start.x
             + deleted.deletedCompo.width; i++)
39      {
40          for (int j = deleted.deletedCompo.start.y; j < deleted.deletedCompo.
                start.y + deleted.deletedCompo.size; j++)
41          {
42              cell(j, i) = deleted.index;
43          }
44      }
45  }
46
47  static void UndoWiring(Wiring wired)
48  {
49      for (int i = 0; i < 256; i++)
50          AlreadyUpdated[i] = false;
51      UpdateChildCount(wired.sender, false);
52      ComponentList[wired.connection.receiver]
53          .inpSrc[wired.connection.receiveIndex] = (Pair){-1, -1};
54      ComponentList[wired.connection.receiver].inputs[wired.connection.
            receiveIndex] = NULL;
55  }
56
57  static void UndoPlacing(Place placed, int *grid)
58  {
59      DeleteComponent(grid, placed.component.start);
60  }
61
62  static void UndoMoving(Move moved, int *grid)
63  {
64      Component compo = ComponentList[moved.index];
65      for (int i = moved.after.x; i < moved.after.x + compo.width; i++)
66          for (int j = moved.after.y; j < moved.after.y + compo.size; j++)
67              cell(j, i) = -1;
68      for (int i = moved.before.x; i < moved.before.x + compo.width; i++)
69          for (int j = moved.before.y; j < moved.before.y + compo.size; j++)
70              cell(j, i) = moved.index;
```

```
    ComponentList[moved.index].start = moved.before;
    SetIOPos(&ComponentList[moved.index]);
}

void Undo(int *grid, int *currentUndoLevel, int totalUndoLevel)
{
    if (*currentUndoLevel >= totalUndoLevel)
        return;
    Actions toUndo = UndoBuffer[*currentUndoLevel];
    switch (toUndo.act)
    {
    case 'd':
        UndoDeletion(toUndo.Action.deleted, grid);
        break;
    case 'w':
        UndoWiring(toUndo.Action.wired);
        break;
    case 'p':
        UndoPlacing(toUndo.Action.placed, grid);
        break;
    case 'm':
        UndoMoving(toUndo.Action.moved, grid);
        break;
    default:
        break;
    }
    *currentUndoLevel += 1;
}
```

Redo is also done similarly except that we first move backwards in the undo buffer (index decreases) and then redo the action using a suitable helper function.

```
static void RedoDeletion(Delete deleted, int *grid)
{
    DeleteComponent(grid, deleted.deletedCompo.start);
}

static void RedoWiring(Wiring wired)
{
    for (int i = 0; i < 256; i++)
        AlreadyUpdated[i] = false;
    UpdateChildCount(wired.sender, true);
```

```
    ComponentList[wired.connection.receiver].inpSrc[wired.connection.
        receiveIndex] = (Pair){wired.sender, wired.connection.sendIndex};
    ComponentList[wired.connection.receiver].inputs[wired.connection.
        receiveIndex] = &ComponentList[wired.sender];
}

static void RedoPlacing(Place placed, int *grid)
{
    Selection placing = {.type = placed.component.type, .size = placed.
        component.inum, .pos = placed.component.start};
    InsertComponent(grid, placing, placed.component.width, placed.component.
        size);
}

static void RedoMoving(Move moved, int *grid)
{
    Component compo = ComponentList[moved.index];
    for (int i = moved.before.x; i < moved.before.x + compo.width; i++)
        for (int j = moved.before.y; j < moved.before.y + compo.size; j++)
            cell(j, i) = -1;
    for (int i = moved.after.x; i < moved.after.x + compo.width; i++)
        for (int j = moved.after.y; j < moved.after.y + compo.size; j++)
            cell(j, i) = moved.index;
    ComponentList[moved.index].start = moved.after;
    SetIOPos(&ComponentList[moved.index]);
}

void Redo(int *grid, int *currentUndoLevel, int totalUndoLevel)
{
    *currentUndoLevel -= 1;
    Actions toRedo = UndoBuffer[*currentUndoLevel];
    switch (toRedo.act)
    {
    case 'd':
        RedoDeletion(toRedo.Action.deleted, grid);
        break;
    case 'w':
        RedoWiring(toRedo.Action.wired);
        break;
    case 'p':
        RedoPlacing(toRedo.Action.placed, grid);
```

```
48          break;
49      case 'm':
50          RedoMoving(toRedo.Action.moved, grid);
51          break;
52      default:
53          break;
54      }
55  }
```

3.4.4 Opening / Saving a file

Opening and Saving file are done by using the `fwrite` and `fread` function provided by the `stdio.h` header. To make this prosess easier for the user we have used the Windows API to open a dialog boc for the user in which they can open an existing file or create a file to save to.

```
1   void ReadFromFile(int *grid, char *fileName)
2   {
3       FILE *data = fopen(fileName, "rb");
4
5       fread(&componentCount, sizeof(unsigned char), 1, data);
6       fread(ComponentList, sizeof(Component), componentCount, data);
7       fread(grid, sizeof(int), GRID_COL * GRID_ROW, data);
8
9       for (int i = 0; i < componentCount; i++)
10      {
11          for (int j = 0; j < ComponentList[i].inum; j++)
12          {
13              ComponentList[i].inputs[j] = &ComponentList[ComponentList[i].inpSrc[
                    j].x];
14          }
15      }
16
17      fclose(data);
18  }
19
20  void SaveToFile(int *grid, char *fileName)
21  {
22      FILE *data = fopen(fileName, "wb");
23      fwrite(&componentCount, sizeof(unsigned char), 1, data);
24      for (int i = 0; i < componentCount; i++)
25      {
```

```
26          fwrite(&ComponentList[i], sizeof(Component), 1, data);
27      }
28      for (int i = 0; i < GRID_ROW * GRID_COL; i++)
29      {
30          fwrite(&grid[i], sizeof(int), 1, data);
31      }
32      fclose(data);
33  }
34
35  void UpdateWindowTitle(char *FileName){
36      char size = 0;
37      char count = 0;
38      char name[50] = "";
39      while (FileName[size] != '\0')
40      {
41          if (FileName[size] == '\\')
42              count = 0;
43          name[count] = FileName[size + 1];
44          count++;
45          size++;
46      }
47
48      char title[70] = "MinimaLogic";
49      SDL_strlcat(title, "-", 70);
50      SDL_strlcat(title, name, 70);
51      SDL_SetWindowTitle(window, title);
52
53  }
54
55  void ChooseFile(int *grid, bool saving)
56  {
57      char FileName[256] = "";
58
59      SDL_SysWMinfo wmInfo;
60      SDL_VERSION(&wmInfo.version);
61      SDL_GetWindowWMInfo(window, &wmInfo);
62
63      OPENFILENAME ofn;
64      memset(&ofn, 0, sizeof(ofn));
65      ofn.lStructSize = sizeof(ofn);
66      ofn.hwndOwner = wmInfo.info.win.window;
```

```
    ofn.hInstance = NULL;
    ofn.lpstrFilter = "Project Files (*.mlg)\0*.mlg";
    ofn.lpstrFile = FileName;
    ofn.nMaxFile = MAX_PATH;
    ofn.lpstrTitle = saving ? "Save File" : "Open File";
    ofn.lpstrDefExt = "mlg";
    ofn.Flags = OFN_NONETWORKBUTTON |
                OFN_FILEMUSTEXIST |
                OFN_HIDEREADONLY;
    if (!saving)
    {
        if (!GetOpenFileName(&ofn))
        {
            return;
        }
        else
        {
            ReadFromFile(grid, FileName);
            fileExists = true;
            SDL_strlcpy(currentFile, FileName, 256);
        }
    }
    else
    {
        if (!GetSaveFileName(&ofn))
        {
            return;
        }
        else
        {
            SaveToFile(grid, FileName);
            fileExists = true;
            SDL_strlcpy(currentFile, FileName, 256);
        }
    }
    UpdateWindowTitle(FileName);
}
```

3.5 Rendering

Note: Throughout this document, the words rendering and drawing have been used interchangeably

There are generally two ways to display graphic elements in an SDL window - displaying images using SDL surfaces or hardware rendering with textures and renderer. This program uses the later as the elements are too versatile to store them as bitmaps.

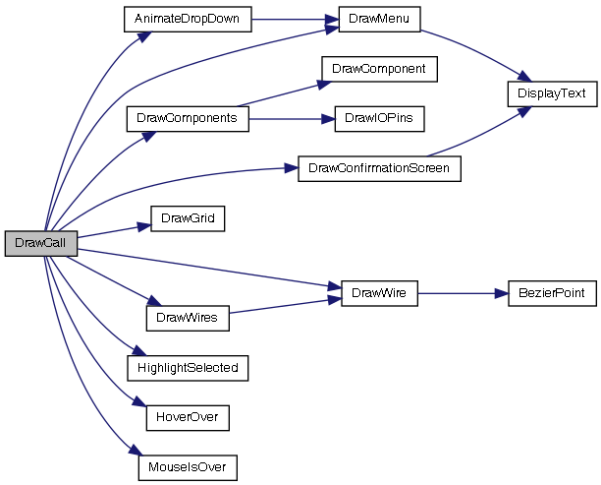

Figure 3.2: Call tree for Rendering

For rendering all elements of the program, many functions have been called as shown in above figure. Most of these functions have been described below.

3.5.1 Functions for rendering in SDL

SDL provides a handful of functions which can be used to draw shapes such as lines, points, rectangles and filled rectangles. These shapes are very simple however the can be utilized to draw other complex shapes. But before we render a shape, we have to set its color. Each element on the screen may have a different color. So the renderer stores the color information when drawing anything on the screen. This information is stored if 4 channels commonly known as rgba format. The first three channels indicate intensity of red, green, and blue color respectively. The last channel is used to store the opaqueness of the element. The higher the value the higher the intensity. Before drawing an element on the screen, we must make sure that the renderer is set to the correct color value. This can be done using the function shown below.

```
1 // SDL_SetRenderDrawColor(SDL_Renderer *, int red, green, blue, alpha);
2 SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255); //opaque white color
```

Rendering Lines

As mentioned previously, SDL provides direct functionality to draw a straight line be-

tween two points. The data doesn't need to be stored anywhere instead we can simply pass the co-ordinates of the points as parameters to the `SDL_RenderDrawLine()` function and it will draw a line between those points for us. The co-ordinates are relative to the top left point of the window. This function has been used to draw the grid lines.

```
// SDL_RenderDrawLine(SDL_Renderer *, startX, startY, endX, endY);
SDL_SetRenderDrawLine(renderer, 0, 0, 50, 50);    //diagonal line form top left
    corner
```

To draw many lines at once, the `SDL_RenderDrawLines()` function can be used which which takes an array of `SDL_Point` and draws may lines by joining n^{th} point in the array to the $(n+1)^{th}$ point. This function has been used to draw wires joining the components.

```
SDL_RenderDrawLines(SDL_Renderer *, SDL_Point *array_of_points, int
   no_of_points);
```

Rendering Rectangles

In exchange for the low level access SDL provides, a lot of abstraction has been taken away. A rectangle is arguably the most complicated shape that can be drawn directly from the renderer in SDL. It can either be a rectangular outline or a solid rectangle filled with the current color configurations. In order to do so, the data for a rectangle must be stored using `SDL_Rect` structure. `SDL_Rect` structure has following members:

```
   int x, y;  // coordinate of top left corner of the rectangle
   int w, h;  // width and height of the rectangle
```

The `SDL_RenderDrawRect()` function can be used to render an outline of a rectangle and `SDL_RenderFillRect()` function can be used to render a filled rectangle. Both of these functions take a `SDL_Renderer *` and a `SDL_Rect *` as parameters.

```
SDL_Rect example = {10, 10, 200, 400};
SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
//SDL_RenderDrawRect (SDL_Renderer *, SDL_Rect *);
SDL_RenderDrawRect (renderer, &example); // draws a red outline of a 200x400
   px rectangle whose top left corner is at 10, 10
//SDL_RenderFillRect (SDL_Renderer *, SDL_Rect *);
SDL_RenderFillRect (renderer, &example); // draws a filled red 200x400 px
   rectangle whose top left corner is at 10, 10
```

The `SDL_RenderDrawRect()` function has been used to draw highlight border around buttons and components. The `SDL_FillRect()` has been heavily used to draw buttons, components, dialog boxes and menus.

In addition to these, `SDL_RenderFillRects()` and `SDL_RenderDrawRects()` are also available which can be used to draw multiple rectangles at once. We have used these functions to draw input/output terminals for the components.

```
SDL_RenderDrawRects(SDL_Renderer *, SDL_Rect *array_of_rect, int no_of_rects);
SDL_RenderFillRects(SDL_Renderer *, SDL_Rect *array_of_rect, int no_of_rects);
```

Rendering Text

As mentioned earlier, the text rendering is handled by an extension of the SDL library - SDL_ttf. It provides the functionality to load ttf, rtf or otf fonts and create textures from them. An array of texture pointers is used to store the necessary ASCII characters. A texture is a structure that stores pixel data that can only be accessed through the GPU. Later, this array is used to display any message we want. In our program, we have implemented the text rendering as follows:

```
static void DisplayText(char *message, SDL_Rect parent){
    /* this function uses the character maps created
       during initialization to render certain text onto the screen */
    char *tmp = message;
    int totalWidth = 0; // total width of the message in pixels
    float factor = 1; // factor to resize the text if it is too long
    // calculate total width
    for (; *tmp; tmp++)
        totalWidth += characterWidth[*tmp - 32];
    // rect onto which text will be rendered
    SDL_Rect charDest = {.y = parent.y, .h = parent.h};
    // calculate the factor
    if (totalWidth > parent.w){
        factor = parent.w / (float)totalWidth;
        tmp = message;
        totalWidth = 0;
        for (; *tmp; tmp++)
            totalWidth += characterWidth[*tmp - 32] * factor;
    }
    // center the text relative to the parent rect
    charDest.x = parent.x + (parent.w - totalWidth) / 2;
    // render each character in message onto the screen
    for (int i = 0; *message; message++, i++){
        charDest.w = characterWidth[*message - 32] * factor;
        SDL_RenderCopy(renderer, characters[*message - 32], NULL, &charDest);
        charDest.x += charDest.w;
    }
}
```

3.5.2 Drawing the Menu

```
static void DrawMenu(bool menuExpanded, bool simulating, bool snap, Selection
    choice){
    // width, height of the window
    int w, h;
    SDL_GetWindowSize(window, &w, &h);
    // background of menu
    SDL_Rect menuBg = {0, 0, MENU_WIDTH, h};
    SDL_SetRenderDrawColor(renderer, BG1);
    SDL_RenderFillRect(renderer, &menuBg);
    // draw RUN/STOP button
    SDL_SetRenderDrawColor(renderer, SideMenu[sm_run].color.r, SideMenu[sm_run
        ].color.g, SideMenu[sm_run].color.b, 255);
    SDL_RenderFillRect(renderer, &SideMenu[sm_run].buttonRect);
    if (simulating)
        DisplayText("STOP", SideMenu[sm_run].buttonRect);
    else
        DisplayText("RUN", SideMenu[sm_run].buttonRect);

    //draw all buttons in the side menu
    SDL_SetRenderDrawColor(renderer, SideMenu[sm_compo].color.r, SideMenu[
        sm_compo].color.g, SideMenu[sm_compo].color.b, 255);
    for (int i = 1; i < sm_total; i++){
        if (i == sm_inc || i == sm_dec && choice.type < g_and)
            continue;
        SDL_RenderFillRect(renderer, &SideMenu[i].buttonRect);
        DisplayText(SideMenuButtonText[i], SideMenu[i].buttonRect);
    }

    if (snap)
        DisplayText("Snap to Grid: On", SideMenu[sm_snap].buttonRect);
    else
        DisplayText("Snap to Grid: Off", SideMenu[sm_snap].buttonRect);

    // draw buttons to inc/dec inputs and also display currnt no. of inputs
    if (choice.type >= g_and){
        SDL_SetRenderDrawColor(renderer, BLACK, 255);
        SDL_RenderFillRect(renderer, &InputsCount);
        char tmptxt[10] = "Inputs: ";
        tmptxt[8] = (char)(choice.size - 2 + 50);
        DisplayText(tmptxt, InputsCount);
```

```
38
39          SDL_SetRenderDrawColor(renderer, SideMenu[sm_inc].color.r, SideMenu[
                sm_inc].color.g,
40                                 SideMenu[sm_inc].color.b, 255);
41          SDL_RenderFillRect(renderer, &SideMenu[sm_inc].buttonRect);
42          DisplayText("+", SideMenu[sm_inc].buttonRect);
43          SDL_RenderFillRect(renderer, &SideMenu[sm_dec].buttonRect);
44          DisplayText("-", SideMenu[sm_dec].buttonRect);
45      }
46
47      if (menuExpanded){
48          SDL_Rect wrapper = {SideMenu[sm_compo].buttonRect.x,
49                              SideMenu[sm_compo].buttonRect.y +
50                              SideMenu[sm_compo].buttonRect.h,
51                              SideMenu[sm_compo].buttonRect.w, 2 + g_total * (25 +
                                    2)};
52          SDL_SetRenderDrawColor(renderer, BG2);
53          SDL_RenderFillRect(renderer, &wrapper);
54
55          for (int i = 0; i < g_total; i++){
56              SDL_SetRenderDrawColor(renderer, Components[i].color.r,
57                                     Components[i].color.g, Components[i].color.b,
                                        255);
58              SDL_RenderFillRect(renderer, &Components[i].buttonRect);
59              DisplayText(compoTexts[i], Components[i].buttonRect);
60          }
61      }
62  }
```

3.5.3 Drawing the Grid

```
1  static void DrawGrid(int pad_x, int pad_y){
2      SDL_SetRenderDrawColor(renderer, BG2);
3      for (int x = 0; x <= GRID_ROW; x += SCALE)
4          SDL_RenderDrawLine(renderer, pad_x + x * CELL_SIZE, pad_y, pad_x + x *
               CELL_SIZE, pad_y + GRID_COL * CELL_SIZE);
5      for (int y = 0; y <= GRID_COL; y += SCALE)
6          SDL_RenderDrawLine(renderer, pad_x + GRID_ROW * CELL_SIZE, pad_y + y *
               CELL_SIZE, pad_x, pad_y + y * CELL_SIZE);
7  }
8
```

```
static void DrawComponents(int pad_x, int pad_y){
    for (int i = 0; i < componentCount; i++){
        if (ComponentList[i].type != probe)
            DrawComponent(ComponentList[i].width, ComponentList[i].size,
                ComponentList[i].start, ComponentList[i].type, pad_x, pad_y, 255,
                 ComponentList[i].outputs[0]);
        else if (ComponentList[i].inpSrc[0].y >= 0)
            DrawComponent(ComponentList[i].width, ComponentList[i].size,
                ComponentList[i].start, ComponentList[i].type, pad_x, pad_y, 255,
                 ComponentList[i].inputs[0]->outputs[ComponentList[i].inpSrc[0].y
                ]);
        else
            DrawComponent(ComponentList[i].width, ComponentList[i].size,
                ComponentList[i].start, ComponentList[i].type, pad_x, pad_y, 255,
                 false);
        if (ComponentList[i].type == d_oct || ComponentList[i].type == d_4x16){
            for (int j = 0; j < ComponentList[i].onum; j++){
                if (ComponentList[i].outputs[j]){
                    SDL_Rect display;
                    display.w = ComponentList[i].width / 2 * CELL_SIZE;
                    display.h = ComponentList[i].size / 2 * CELL_SIZE;
                    display.x = ComponentList[i].start.x * CELL_SIZE + pad_x +
                        ComponentList[i].width / 4 * CELL_SIZE;
                    display.y = ComponentList[i].start.y * CELL_SIZE + pad_y +
                        ComponentList[i].size / 4 * CELL_SIZE;
                    SDL_RenderCopy(renderer, displayChars[j], NULL, &display);
                    break;
                }
            }
        }
        DrawIOPins(ComponentList[i], pad_x, pad_y);
    }
}
```

3.5.4 Drawing Wires

Drawing straight lines as wires would clog up the canvas and the circuits would look untidy, so a cubic Bezier curve is used to draw wires. A Bezier curve uses a set of fixed anchor points to draw a curve of certain order. The algorithm to trace such a curve comprises of nested linear interpolation (lerp). The level of nesting is determined by the degree of the curve. Here, is an example of a Bezier curve.

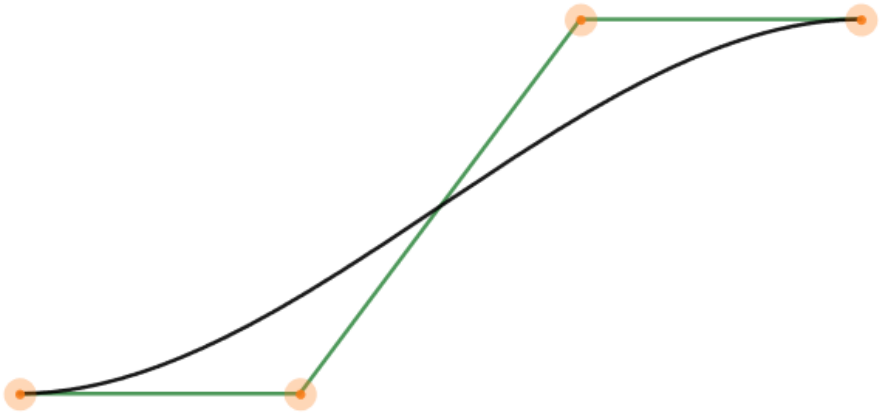

Figure 3.3: Bezier Curve

This is implemented in our code as follows:

```
// p[4] are the four anchor points
static SDL_Point BezierPoint(float t, SDL_Point p[4]){
    float tt = t * t;
    float ttt = tt * t;
    float u = 1 - t;
    float uu = u * u;
    float uuu = uu * u;

        //returns a point on the curve for a certain value of t
    return (SDL_Point){
       uuu * p[0].x + 3 * uu * t * p[1].x + 3 * u * tt * p[2].x + ttt * p[3].x,
       uuu * p[0].y + 3 * uu * t * p[1].y + 3 * u * tt * p[2].y + ttt * p[3].y
          };
}
/* Changing the value of t from 0 to 1 in 100 steps will give 100 points on
    the curve. We can draw lines between consecutive points to get the desired
    curve.*/
```

The above function only returns one point which lies on the curve. Curve cannot be drawn based on a single point. So the following function is called which calculates a bunch of points along the curve and joins them with lines to draw a curve:

```
static void DrawWire(SDL_Point start, SDL_Point end, bool hilo, bool
    simulating){
    /*
        This function draws a 3px thick bezier curve (the wire) between points
            start and end
```

```
        this is done by drawing 3 bezier curves next to each other
        hilo boolean represents what signal the wire is carrying (high or low)
        simulating boolean represents whether simulation is running or not
    */
    SDL_Point wirePoints[MAX_WIRE_PTS]; // array of points which will be joined
         to make the wire
    for (int i = 0; i < 3; i++){
        // to align the starting and ending points
        if (abs(start.x - end.x) > abs(start.y - end.y)){
            start.y++;
            end.y++;
        }
        else{
            start.x++;
            end.x++;
        }
        // anchor points between start and end
        SDL_Point p2 = {start.x + (end.x - start.x) / 3, start.y};
        SDL_Point p3 = {end.x - (end.x - start.x) / 3, end.y};
        // setting color of the wire
        // set red color if hilo is true and simulation is running
        // set blue color if hilo is false and simulation is running
        // set green color simulation is not running
        if (i == 1){
            // to give bezier curve in the middle a lighter color
            if (hilo && simulating)
                SDL_SetRenderDrawColor(renderer, HIGH_COLOR, 255);
            else if (!hilo && simulating)
                SDL_SetRenderDrawColor(renderer, LOW_COLOR, 255);
            else
                SDL_SetRenderDrawColor(renderer, WIRE_NEUTRAL, 255);
        }
        else{
            // to give bezier curve in the middle a darker color
            if (hilo && simulating)
                SDL_SetRenderDrawColor(renderer, WIRE_HIGH_D, 255);
            else if (!hilo && simulating)
                SDL_SetRenderDrawColor(renderer, WIRE_LOW_D, 255);
            else
                SDL_SetRenderDrawColor(renderer, WIRE_NEUTRAL_D, 255);
        }
```

```
        // calculate all points along the curve
        for (int i = 0; i < MAX_WIRE_PTS; i++){
            float t = (float)i / MAX_WIRE_PTS;
            wirePoints[i] = BezierPoint(t, (SDL_Point[4]){start, p2, p3, end});
        }
        // make sure that wire touches starting and ending points
        wirePoints[0] = start;
        wirePoints[MAX_WIRE_PTS - 1] = end;
        // join all the points with lines
        SDL_RenderDrawLines(renderer, wirePoints, MAX_WIRE_PTS);
    }
}
```

3.6 Closing

Closing consists of destroying the textures and closing the libraries. This is done by following functions:

```
static void DestroyTextures()
{
    for (int i = 32; i < 127; i++)
        SDL_DestroyTexture(characters[i - 32]);
    for (int i = 0; i < 16; i++)
        SDL_DestroyTexture(displayChars[i]);
}

void CloseEverything()
{
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    DestroyTextures();
    SDL_Quit();
}
```

Chapter 4

RESULT

Figure 4.1: Initial / Starting Screen

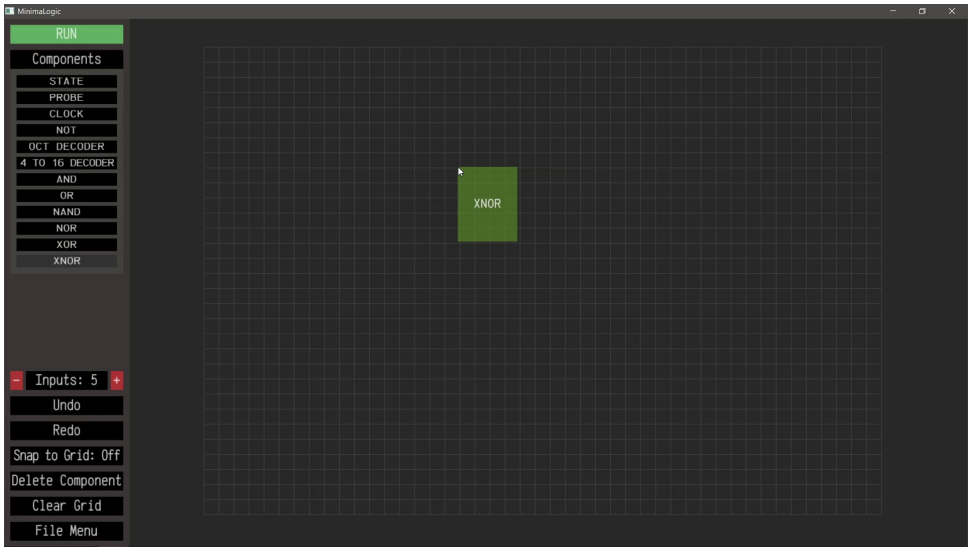

Figure 4.2: Component preview before placing

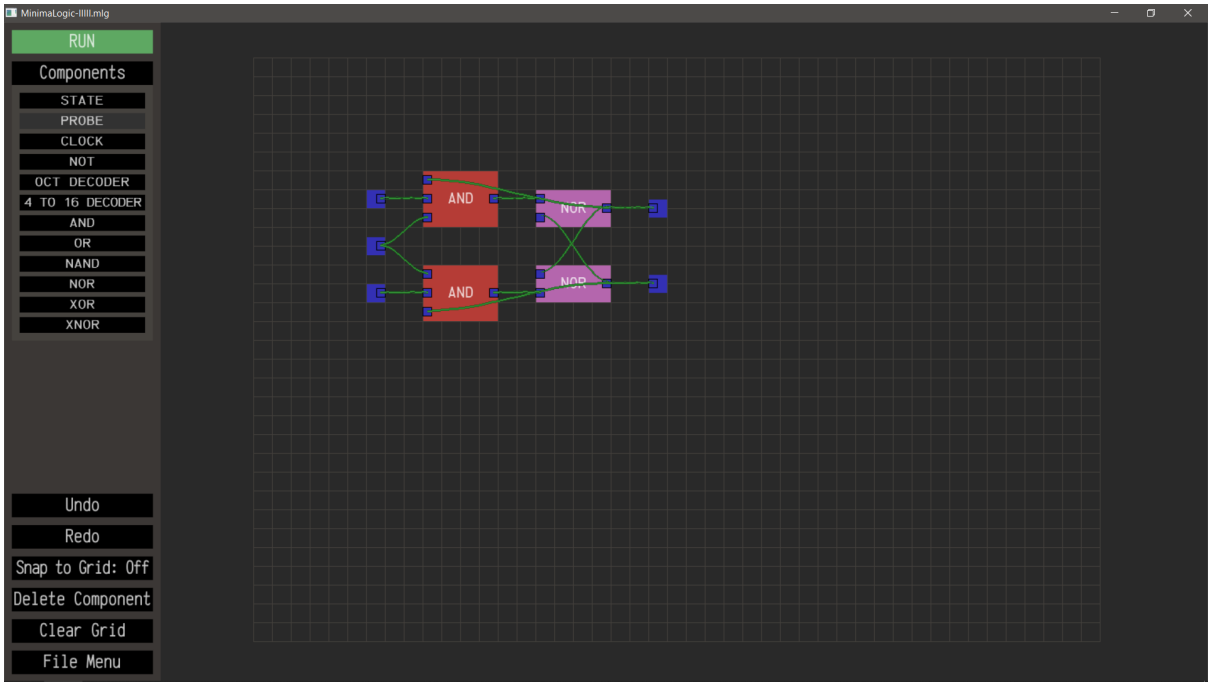

Figure 4.3: Circuit in normal state (not simulated)

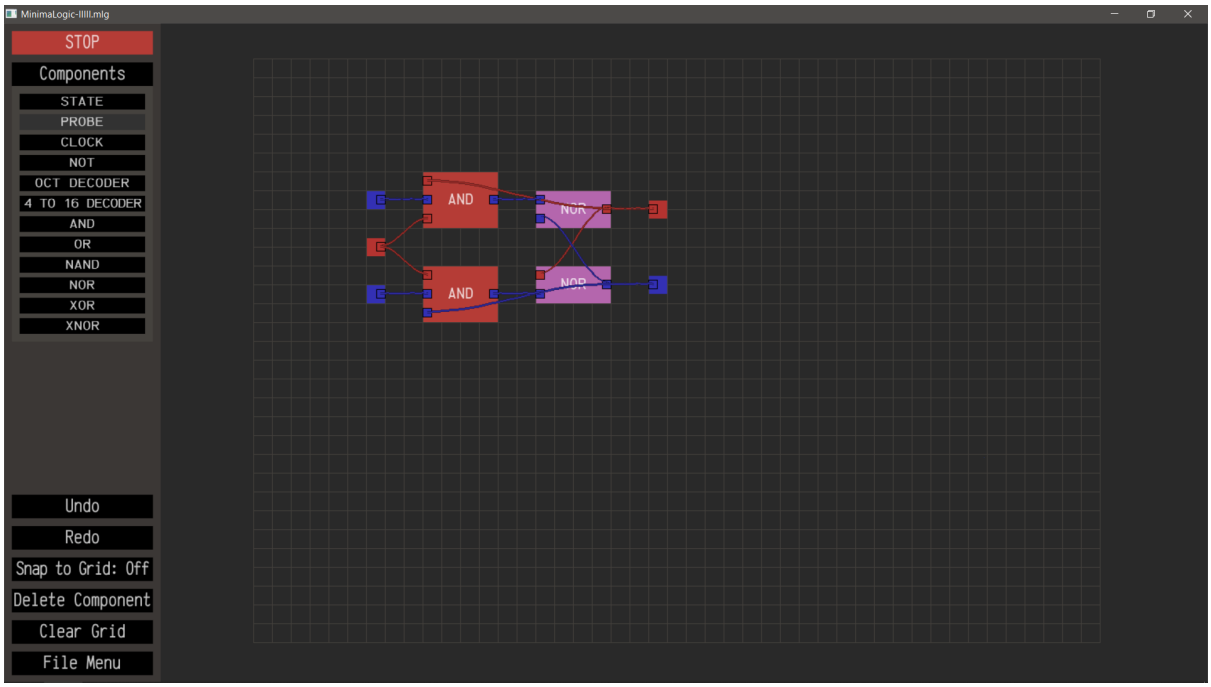

Figure 4.4: Circuit being simulated

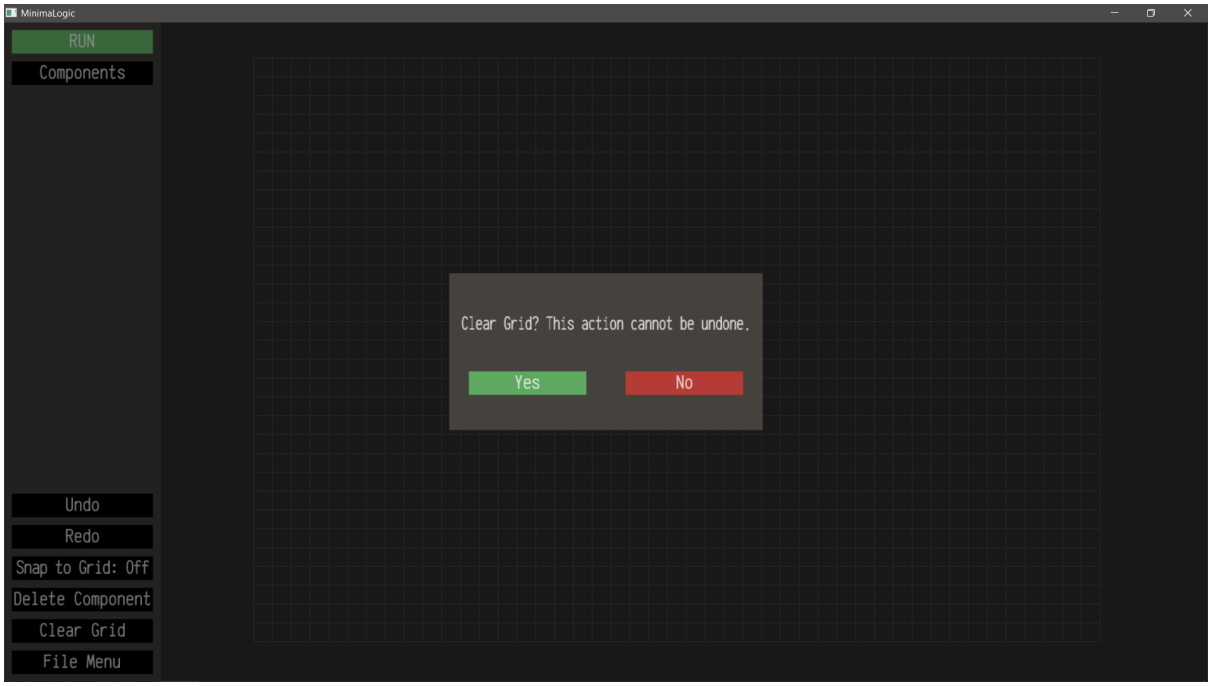

Figure 4.5: Confirmation screen before clearing

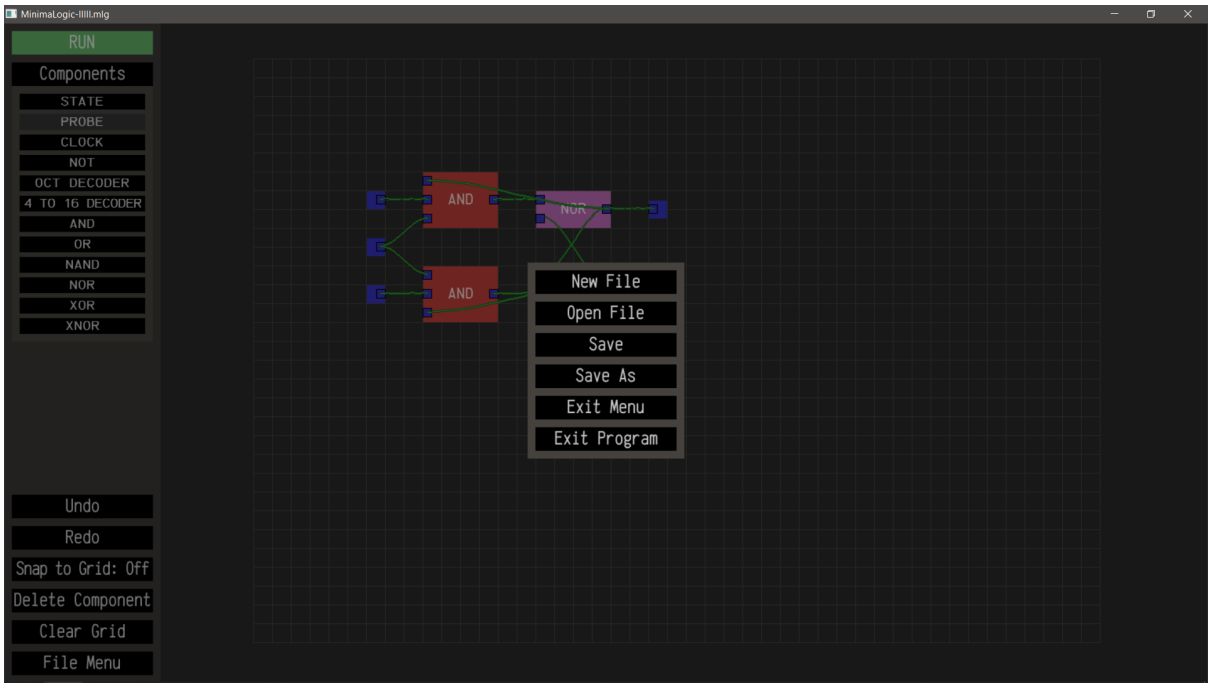

Figure 4.6: File Menu

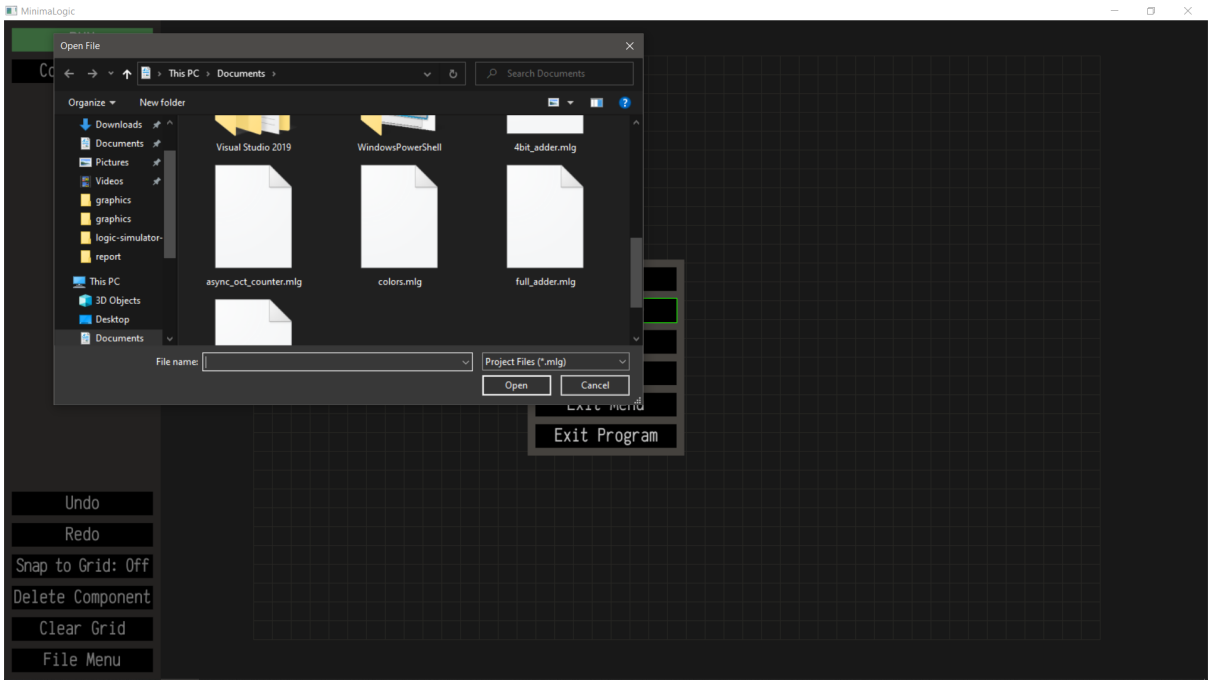

Figure 4.7: Opening a file

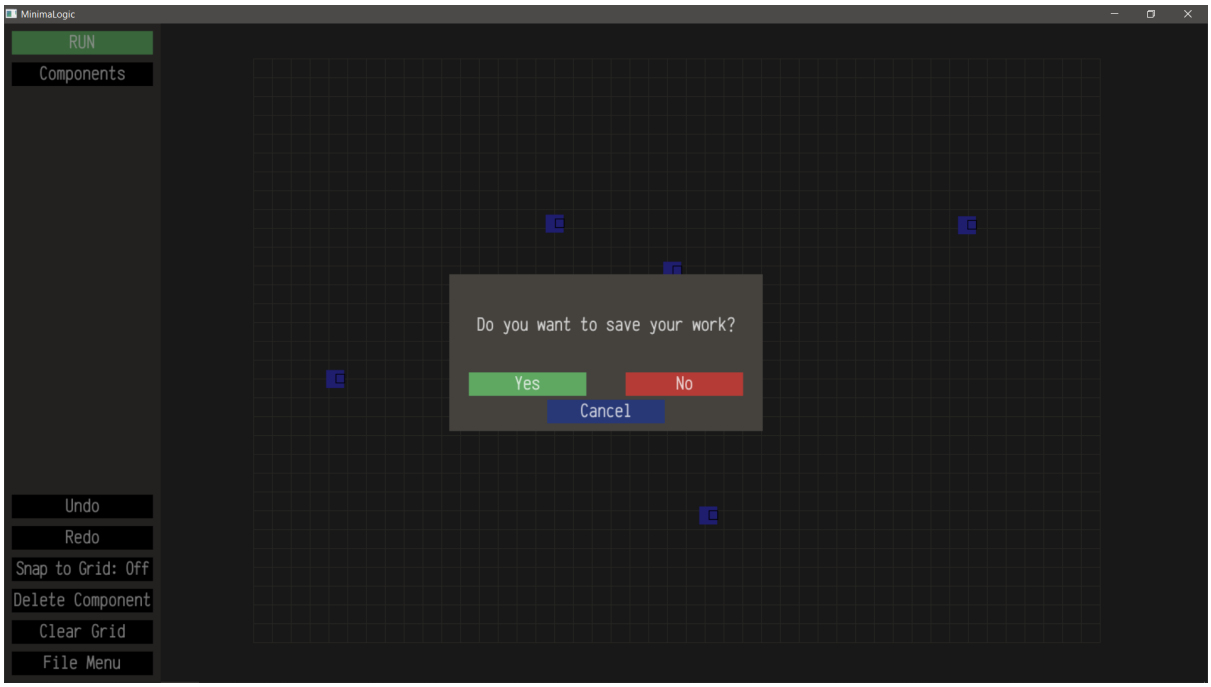

Figure 4.8: Asking to save before exiting program or opening another file

Chapter 5

CONCLUSION

5.1 Experience

When we started working on this project, we were sure it would be an interesting experience, but what we did not anticipate was the sheer number of challenges that would come along. Challenges that proved the knowledge gained from our course to be insufficient for a project of this level. Algorithms that were unfamiliar to us had to be used for various parts of the program. But, along the way, we learned that these challenges were part of the learning experience. All in all, this was a very helpful project that introduced us to different paradigms of programming. As futile as it may be, we've tried to make a list of what we learned by doing this project:

1. Organizing the project and making the source code readable by using multiple files and using a consistent naming convention.
2. Collaborative work with VCS like Git and remote hosting services like GitHub.
3. Using the official documentation of Libraries and the C language itself.
4. Unique algorithms for things like drawing curves, collision detection, etc.
5. Avoiding memory leaks and other vulnerabilities that are exposed in low level language such as C.
6. Making sure the codes were methodical, operational and compatible.

5.2 Overview of the project

A one sentence definition of this project could be : "An optimal use of C programming language, learnt throughout the semester, to create a Logics Simulator which will get the basic tasks done." We can summarise the overall features of the project in these points:

1. Features digital electronics components required to make the logic circuits.
2. User friendly interactive approach to learning the basics.
3. Allows the modification and analysing of the tasks as required by the user.
4. Incorporates an engaging blueprint and layout.

5.3 Possible Improvements

While this program is stable as far as we have tested it, it is nowhere near perfect. Many features were sratched off of the to-to list, some due to the lack of time and others because they were too complicated. Here is a list of improvements that can be made to this program.

1. Better graphics for components and wires. The wires look jagged right now which can be fixed with some anti-aliasing techniques.
2. Selection of multiple components on the grid and the functionality to copy paste the selection.
3. Zooming and panning to provide a larger canvas.
4. More components. Currentlty, there are very few, basic components which makes drawing more complex circuits difficult.
5. Letting users create custom components.