PaintBFS: Paint Program with Shortest Path Implementation

Loigen Sodian, Isaac Koo Hern En

May 2024

1 Introduction

The design being presented is a paint program that allows users to draw on a screen with up to 64 different colors to choose from on a 512 by 480 screen, divided into 8 by 8 tiles (a total of 3840 tiles). Users can also designate a starting and ending point on the screen, and the program will find and display the shortest path between these points using the Dijkstra algorithm, effectively implemented as a Breadth-First Search (BFS). The implemented algorithm is very flexible with path findings, it can navigate through obstacles and can detect if no path is found.

The operation of the program is straightforward; the user presses the left mouse button and drag the mouse around to draw on the screen, and is able to cycle through all of the available colors by pressing `Enter`, and can wish to erase the screen by right-clicking, and drag the mouse around the screen.

1.1 Why use BFS for the Shortest-Path?

BFS is used here since the distance between each pixel is the same. Dijkstra Algorithm actually converge to BFS if the weight of the nodes are all equal in distance. Hence, it is still correct to mention that the path-finding algorithm used in this program will still lead to the shortest path.

1.2 Overview of the Circuit

1.2.1 Purpose of the Circuit

The primary purpose of the circuit is to implement an interactive paint program that allows users to draw on a screen and find the shortest path between two designated points. This is achieved by integrating a user interface for drawing with an algorithmic core that finds the shortest path between the two pixels.

1.2.2 Features

The features are summarized as follows:

- **Drawing Capability**: Users can draw on a 512x480 pixel screen using up to 64 different colors with just a PS/2 Mouse with full cursor support.
- **Color-Aware Cursor**: Users can be aware of the colors they are drawing, as the cursor color changes to the color that the palette is choosing at that instant.
- **Shortest-path built-in**: Users can designate a start and end point on the drawing, and the system will compute and display the shortest path using the Dijkstra algorithm, implemented as a Breadth-First Search (BFS). The algorithm recognizes obstacles, and can detect impossible paths (i.e., no valid paths).

1.3 Inputs and Outputs of the Circuit

The inputs and outputs are listed as follows:

1.3.1 Input of the Circuit

- **Keyboard**: User can either presses `1`, which will draw the color used to determine the starting point (`C0C0C0`), `2` to determine the ending point (), and `Enter` to cycle through the other 62 colors.

- **Mouse**: User can either left-click and drag to draw on the screen with the selected color, or right-click and drag to erase the colors on the screen.
- **Pushbuttons**: Pushbutton KEY[1] serves as the ”Start” button of the shortest-path algorithm. When valid, pressing this button starts the whole operation.

1.3.2 Output of the Circuit

- **VGA Screen**: The output of the drawing (and the path-finding) will be shown real-time on the screen.
- **7-segment Display**: The 7-segment display shows the current color being drawn, as described in `Palettes.sv`. (e.g., 0 being Black, 1 being the starting point's color, and 2 being the ending point's color)
- **LED Output**: The LED light shows the mode of operation, such as the ”valid” light, which turns on if the starting and ending point has been drawn (so that the shortest-path can be drawn), and other control lights (i.e., the ”done” and the ”Run” signal) that serve to debug the circuit.

1.3.3 How I/O are processed

The input keyboard is connected through the NIOS interface, firstly going through the EZ-OTG controller and then HPI I/O Interface. The output keycode is a value that correspond to each key press (e.g., 4 being associated with ”A”, 5 being associated with ”B”, and so on). The only key codes that are concerned are the buttons ”1” (30), ”2” (31), and ”Enter (40). The selected color will then be passed to `PixelRender.sv`. This module converts the value into the value that corresponds to the Palette we designed, and this value is used to write on the screen (if its being written) and to the 7-segment display (which shows the current selected color).

The mouse is connected through the PS/2 Mouse controller, with the code provided by Altera in the DE2-115 CD-ROM. The output of this controller is an 8-bit value that is (unfortunately) not enough to cover the whole screen. As a fix, we zero-extend this value to 16-bits, and then multiply the 8-bit value by 4. This way, drawing the 8 by 8 tile will still be possible and easy (since 2 pixels are mapped to one tile, which reduces the possibility of missing a tile when drawing a straight line), while also allow us to cover the whole 512 by 480 screen. The new values (`WriteX`, `WriteY`) are then loaded to `PixelRender.sv`, where if the user also presses the left (or right) mouse button, then `PixelRender` will save the corresponding tile (calculated as `WriteX + (WriteY << 6)`) with the currently selected color on the screen buffer, which will draw be drawn on the screen. The right-click operation is the same, except that the color being written is black (to give the effect of erasing the screen, since the screen base is black). If no drawing is being done, then the position of the cursor is only used to draw the cursor with the color that is currently being selected, all managed in `PixelRender.sv`.

The KEY[1] push-button acts as the starting switch of the algorithm. If the user presses this button, the machine will go to a ”checking” state where it checks if both the starting and ending point is defined, for which it will then start the path-finding algorithm in `DijkstraCore.sv`.

The LEDs are tied to the control signals of the circuit. Currently there are four signals tied to the LED, `done` (which indicate if the algorithm has finished running), `check` (KEY[1]), `valid` (turns on only if both the starting and ending point is defined), `Run` (turns on when the path-finding algorithm is running).

1.3.4 Output that is Shown

The main output is shown on a VGA display, where users can see their drawings in real-time. The start and end points are highlighted, and the computed shortest path is drawn over the existing drawing. The display dynamically updates based on user interactions and algorithmic computations. User also can see the color being selected through the mouse cursor and also the hex-display on the FPGA.

2 Module Description

Note that the use of `wire` indicate that it is an `inout` connection.

2.1 `toplevel`

2.1.1 Inputs

- `logic CLOCK_50`
- `logic [3:0] KEY`

- wire PS2_CLK
- wire PS2_DAT
- wire PS2_CLK2
- wire PS2_DAT2
- wire [15:0] OTG_DATA
- wire [31:0] DRAM_DQ

2.1.2 Outputs

- logic [6:0] HEX0
- logic [6:0] HEX1
- logic [8:0] LEDG
- logic [17:0] LEDR
- wire PS2_CLK
- wire PS2_DAT
- wire PS2_CLK2
- wire PS2_DAT2
- logic VGA_CLK
- logic VGA_SYNC_N
- logic VGA_BLANK_N
- logic VGA_VS
- logic VGA_HS
- logic [7:0] VGA_R
- logic [7:0] VGA_G
- logic [7:0] VGA_B
- wire [15:0] OTG_DATA
- logic [1:0] OTG_ADDR
- logic OTG_CS_N
- logic OTG_RD_N
- logic OTG_WR_N
- logic OTG_RST_N
- logic OTG_INT
- logic [12:0] DRAM_ADDR
- wire [31:0] DRAM_DQ
- logic [1:0] DRAM_BA
- logic [3:0] DRAM_DQM
- logic DRAM_RAS_N
- logic DRAM_CAS_N

- `logic DRAM_CKE`
- `logic DRAM_WE_N`
- `logic DRAM_CS_N`
- `logic DRAM_CLK`

2.1.3 Description and Purpose

Serves as the top-level entity of the project. It interfaces between hardware inputs and outputs with the Verilog design (and SoC) to power these hardware as according to the logic output of the modules.

2.2 `DijkstraCore`

2.2.1 Inputs

- `logic Clk`
- `logic Reset`
- `logic Run`
- `logic VGA_CLK`
- `logic [12:0] StartPoint`
- `logic [12:0] EndPoint`
- `logic [31:0] AVL_READDATA`
- `logic AVL_WAIT_REQUEST`

2.2.2 Outputs

- `done`
- `AVL_READ`
- `AVL_WRITE`
- `AVL_CS`
- `AVL_BYTE_EN`
- `AVL_ADDRESS`
- `AVL_WRITEDATA`

2.2.3 Description and Purpose

This module serves to determine the shortest distance between the start and end points. Since the distance of the pixels are all the same, then the Dijkstra Algorithm will converge into the BFS algorithm. So in actuality we are using the BFS algorithm to find the path. This module consists of 22 states:

1. RESET
2. IDLE
3. INIT
4. LOAD_END
5. BOUND_CHECK1A
6. BOUND_CHECK1B
7. BOUND_CHECK2A
8. BOUND_CHECK2B
9. BOUND_CHECK3A
10. BOUND_CHECK3B
11. BOUND_CHECK4A
12. BOUND_CHECK4B
13. VISIT_NODES1
14. VISIT_NODES2
15. VISIT_NODES3
16. VISIT_NODES4
17. CHECK_NEIGHBORS
18. DONE_BFS
19. LOAD_PREV1
20. LOAD_PREV2,
21. TRACE_PATH
22. FINISH

In the RESET state, the queue FIFO and the OCM (DijkstraOCM) used to store the previous node, as well as the visited node (visitedOCM), is cleared. Note that both DijkstraOCM and visitedOCM as well as the queue FIFO uses the $x + y \cdot 64$ addressing (or `x + y << 6`). This means the address of a node at $(0, 1)$ will be addressed as node No.64, and its potential neighbors are node No.0, No.65, and No.128.

The IDLE state does basically nothing, until the user presses Start. In this case, the Control unit will send a `Run=1` signal to this module, and the state switches to INIT. In the INIT state, it marks the endpoint node (we start from the endpoint since this way the route tracing will start from the startpoint node) as visited, store itself as the previous node, and pushes it into the queue. LOAD_END dequeues the queue (now the popped node is referred to as `pop_node`), and at the BOUND_CHECK states, all four neighbors of `pop_node` will be checked. The module will initially check if `pop_node` is on the most-top, most-right, most-bottom, or most-left. For example, BOUND_CHECK1A and BOUND_CHECK1B checks if `pop_node` has a upper neighbor or not. If yes, then it will send a read request with the address being this upper node through the Avalon MM bus to Pixel Render's PixelOCM, where the data being read (`AVL_READDATA`) will be routed back to this module and checked. Since we only want to traverse through valid paths, and also want to detect start and end nodes, then if the read data of the pixel is equal to 2 or less (which means the node is either an endpoint: 2, startpoint: 1, or a path: 0), the module will queue this upper neighbor to the queue. Similarly, it will check for the right neighbor in BOUND_CHECK2A and BOUND_CHECK2B, bottom neighbor in BOUND_CHECK3A and BOUND_CHECK3B, and left neighbor at BOUND_CHECK4A and BOUND_CHECK4B. Note that valid neighbor nodes will be written into the `neighbors` register. This register is a 13-bit register, for which the first 12 bits store the coordinate data, and the last bit indicate if this neighbor is valid or not.

VISIT_NODES1, VISIT_NODES2, VISIT_NODES3, VISIT_NODES4 states marks the top, right, bottom, and left node as visited respectively, and also store `pop_node`'s node value in the addresses of each of these four potential nodes. Note that if one of the neighboring nodes are the starting node (i.e., the destination), then the module will raise a "found end" signal, which will jump the state directly to `DONE_BFS`. At VISIT_NODES4, the module will check if the queue is empty. If so, then a "dead end" signal will be raised and the state will directly jump to FINISH. Otherwise, these states also queues these four nodes, for which at CHECK_NEIGHBORS, a node will be dequeued from the queue FIFO, and then it goes back to the BOUND_CHECK1A state.

At DONE_BFS, the module will start backtracing the path back from the starting point to the ending point, by first loading the endpoint node in this state. It then request a read to `DijkstraOCM` to determine the previous node in LOAD_PREV1, where after one clock cycle (i.e., LOAD_PREV2), the previous node will be available at the output of this OCM. The output data will then be sent through the Avalon MM bus back to Pixel Render's PixelOCM to write the path data (with a palette colour of 3) into this node. From which, the state will go back to LOAD_PREV1 again to load the previous node of this node.

In LOAD_PREV2, if the data obtained from DijkstraOCM is the endpoint, then the state will jump to FINISH, where the done flag is asserted, from which the Control will inhibit this module and allow drawing again.

2.3 `Control`

2.3.1 Inputs:

- `logic Clk`
- `logic Reset`
- `logic check`
- `logic valid`
- `logic done`

2.3.2 Outputs:

- `logic Run`

2.3.3 Description and Purpose:

This module serves to control signals used for drawing and path prediction. It has 5 main states, `DRAW`, `VERIFY`, and `PROCESS`. In the DRAW phase, the Dijkstra Core module is inhibited, while the Pixel Render module is able to draw freely and also receive drawing inputs from the mouse. When the user presses `check` (`KEY[1]`), the state will change to VERIFY, where it will check if there is a starting and ending point in the image. If all is good, then the state will change to PROCESS, in which the screen will turn black (since `PixelRender` is being inhibited from

running), and Dijkstra Core will start and determine any shortest path. Once a shortest path is found (or it cannot be determined), the Dijkstra Core module will send a "done" signal to the Control module, in which the state will turn to `DRAW`, where the user can see the path being drawn. The user can now freely draw again.

2.4 `PixelRender`

2.4.1 Inputs:

- `logic Clk`
- `logic Reset`
- `logic blank`
- `logic Run`
- `logic VGA_CLK`
- `logic WE`
- `logic RE_OCM`
- `logic [7:0] WritePixel`
- `logic [9:0] WriteX`
- `logic [9:0] WriteY`
- `logic [9:0] DrawX`
- `logic [9:0] DrawY`
- `logic AVL_READ`
- `logic AVL_WRITE`
- `logic AVL_CS`
- `logic [3:0] AVL_BYTE_EN`
- `logic [31:0] AVL_WRITEDATA`
- `logic [9:0] AVL_ADDRESS`

2.4.2 Outputs:

- `logic [7:0] VGA_R`
- `logic [7:0] VGA_G`
- `logic [7:0] VGA_B`
- `logic [12:0] StartPoint`
- `logic [12:0] EndPoint`
- `logic AVL_WAIT_REQUEST`
- `logic [31:0] AVL_READDATA`

2.4.3 Description and Purpose:

This module serves to draw pixel data to the VGA screen, and also receive simultaneous drawing input from the user. There will be two distinct memories, the `PixelOCM` which is a RAM using the on-chip memory, and also Palette, which is a ROM that supports 64 different colors. in the DRAW phase, this module will receive the x and y coordinate of the drawing location, and will convert this into an 8 by 8 pixel block. This is done by division of 8 (or right shift three times). Then, this value will be converted into the $x + y \cdot 64$ format by left-shifting the y position 6 times, and then adding it up with x. This in the end, represents each pixel with a unique address `readtemp` (e.g., a pixel at (0,1) will have an address at 64). Now, since PixelOCM stores each four pixel data in one address, then we will need to right shift this address (Which is used in the DijkstraCore module) by twice (or division by 4) to obtain the address of the PixelOCM that we will read from `read_addr`. From here, `read_addr` will be passed to PixelOCM, and a 32-bit output will be read from this RAM. Now, since each colors are 8 bit in length, we need to do a modulo division $(x + y \cdot 64)$ % 4, which can be done by simply slicing the first two bits of `readtemp`, i.e., `readtemp[1:0]`. Note that drawing only happens in the 512 by 480 area, and anything beyond will be rendered as black.

Since PixelOCM is a true dualport RAM, it allows simultaneous writing into the memory, which is done by user controlled mouse clicks. When the user presses the mouse and starts drawing, Pixel Render will receive a write request (`WE`), along with the color data, and the x and y position. As long as the position is within 512 by 480, then `WE` will be enabled and Pixel Render will draw. While the control state is at DRAW, then users will be able to draw freely on the monitor, however when the path finding algorithm is running, then writing access will be given to the Dijkstra Core module instead, so users will not be able to draw at this stage.

If the user wishes to draw the start and end points, this module will store the most recent start and end points that the user draw, and mark it as valid. This way, when the user wants to run the path finding algorithm, the control unit will be able to see that both the start and end points are defined. The following are a snipplet of the first 16 colors that the user can choose from the Palettes ROM:

0. Path (Black)
1. Start
2. Finish
3. Pathfinding's path
4. Blue
5. Dark Blue
6. Purple
7. Dark Purple
8. Red
9. Bright Red
10. Pink
11. Yellow
12. Orange
13. Green
14. Light Green
15. Teal
16. Dark Teal

2.5 `Palette`

2.5.1 Inputs:

- `logic VGA_CLK`
- `logic RE_OCM`
- `logic blank`
- `logic [7:0] PixelColor`

2.5.2 Outputs:

- `logic [7:0] VGA_R`
- `logic [7:0] VGA_G`
- `logic [7:0] VGA_B`

2.5.3 Description and Purpose:

This module is a ROM that serves as a LUT for `PixelRender`. `PixelRender` will feed this module with colors defined as `PixelColor` that allows the selection of up to 64 colors. Depending on the selected colors, it will output the R, G, and B values for the pixel being drawn.

In case of blanking (and also when the screen is not being drawn, such as when running the path-finding algorithm), the screen will inhibit any color input and will produce a black screen until the blanking finishes, or until the path-finding algorithm finishes.

2.6 `HexDriver` (Provided by Course)

2.6.1 Input:

- `[3:0] In0`

2.6.2 Output:

- `[3:0] Out0`

2.6.3 Description and Purpose:

This module accepts a value of 4-bit input (i.e., `0` to `F`) and output a signal such that the 7-segment display will display this values. In the case for this project, the Hex drivers will accept the selected color value and prints it out to the hex display for the user to see.

2.7 `hpi_io_intf` (Provided by Course)

2.7.1 Inputs:

- `logic Clk`
- `logic Reset`
- `logic [1:0] from_sw_address`
- `logic [15:0] from_sw_data_out`
- `logic from_sw_r`
- `logic from_sw_w`
- `logic from_sw_cs`
- `logic from_sw_reset`
- `wire [15:0] OTG_DATA`

2.7.2 Outputs:

- `wire [15:0] OTG_DATA`
- `logic [15:0] from_sw_data_in`
- `logic [1:0] OTG_ADDR`
- `logic OTG_RD_N`
- `logic OTG_WR_N`
- `logic OTG_CS_N`
- `logic OTG_RST_N`

2.7.3 Description and Purpose:

This module interfaces between the NIOS processor with the HPI registers. It acts as a tristate, which means that it will write high impedance to `OTG_DATA` if writing operation (`from_sw_w`) is not turned on, or "high." Otherwise, the data out buffer's data will be written to `OTG_DATA`. It also acts as a buffer since `from_sw_data_out` (which is an `inout wire`) should be driven by a register, not combinational logic. The buffer functions to carry on the values of the "`from_sw`" logic to the OTG wire for use in the CY7C67200 chip.

2.8 ps2 (Provided by Altera)

2.8.1 Inputs:

- `logic iSTART`
- `logic iRST_n`
- `logic iCLK_50`
- `wire PS2_CLK`
- `wire PS2_DAT`

2.8.2 Outputs:

- `wire PS2_CLK`
- `wire PS2_DAT`
- `oLEFBUT`
- `oRIGBUT`
- `oMIDBUT`
- `[7:0] mx`
- `[7:0] my`

2.8.3 Description and Purpose:

This module serves to interface the PS/2 Mouse with the rest of the module. It accepts the mouse data from both the PS/2 Clock and data, and converts it to 8-bit x and y coordinate data for further processing.

3 Design Procedure and State Diagram

3.1 Design Procedure

Most of the codes came from either our own creation (such as `toplevel.sv`, `PixelRender.sv`, `DijkstraCore.sv`, `Palettes.sv`, and `Control.sv`), or provided codes from earlier experiment classes taken in ECE385 (keyboard HPI I/O Interface and EZ-OTG chip along with the software codes and 7-segment display driver). However, we also have taken the code for the PS/2 Mouse from Intel Altera's CD-ROM for DE2-115, which contains a demo for PS/2 Driver.

The objective of this design is to have an easy input from the user through both the mouse and keyboard (which means zero to minimal driver using the FPGA's push buttons or switches) that can in real-time, update the screen with drawings, and also conduct path-finding algorithm that is fast and accurate, based on the user-given constraints (such as obstacles).

Research is primarily done in the algorithm of BFS, and also on timings for the on-chip memory and FIFO during read or write operations. We also intensively study the use of Avalon MM Master and Slave to integrate it to our project.

The design is composed of two objectives, a paint program and a path-finder machine. We first build the system to draw pixels, and then some of its properties (such as the addressing standard we are using, i.e., `x+y<<6`) to find the shortest path between the two nodes.

3.2 State Diagram

Figure 1 shows the state diagram of the design. There is two control states, one handled by `Control.sv`, another by `DijkstraCore.sv`.

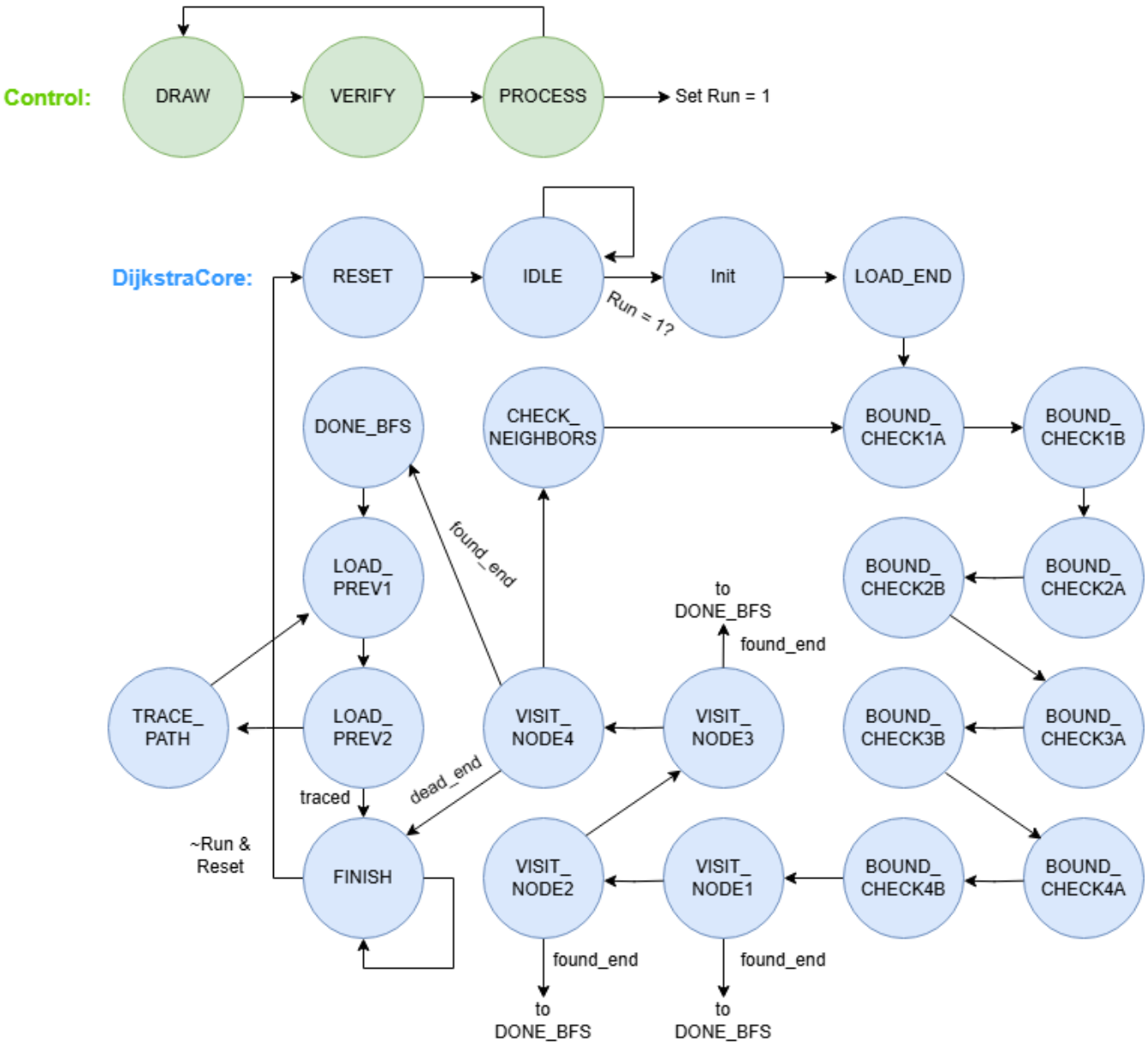

Figure 1: State diagram of the whole design. Green is the state within `Control`, while Blue is the state within `DijkstraCore`.

4 Block Diagram

Figure 2 shows the block diagram:

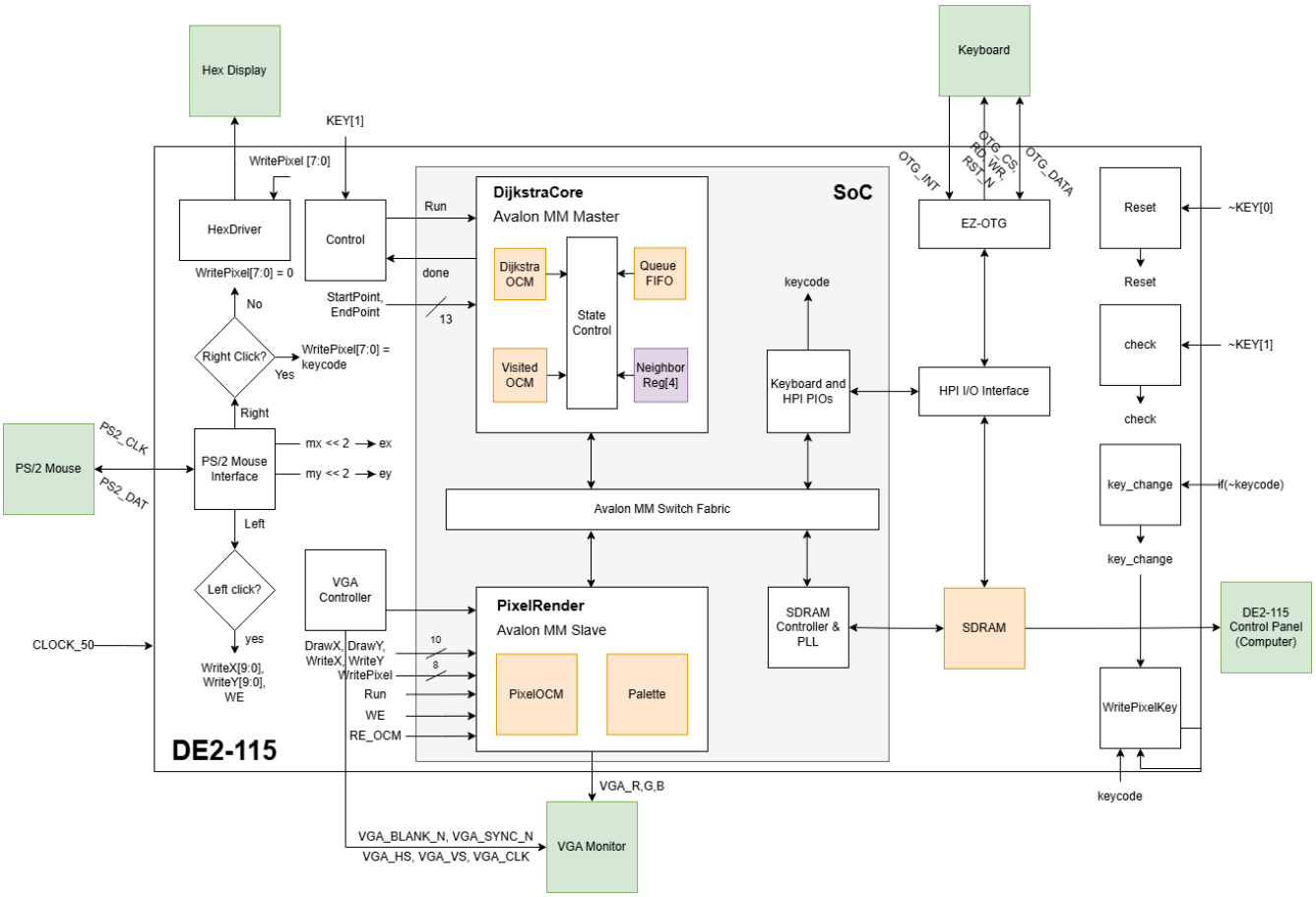

Figure 2: Block Diagram. Green represents external connections, orange indicate RAMs, ROMs, or FIFO, while purple indicate registers

On the Avalon MM bus, the NIOS II softcore processor, PIOs for the keyboard, the SDRAM Controller and its PLL clock, the Dijkstra Core Avalon MM Master IP Core, and the Pixel Render Avalon MM Slave IP Core. The addressing is as follows:

No	Components	Starting Address	Ending Address
1	On-chip Memory RAM/ROM	s1: x0000, s2: x8000	s1: x3fff, s2: xffff
2	SDRAM Controller	x1000_0000	x17ff_ffff
3	SDRAM PLL	x0000_d080	x0000_d08f
4	System ID Peripheral	x0000_d0a0	x0000_d0a7
5	NIOS II Processor	x0000_c800	x0000_cfff
6	JTAG UART	x0000_d0a8	x0000_d0af
7	Keycode PIO	x0000_d070	x0000_d07f
8	HPI Read PIO	x0000_d040	x0000_d04f
9	HPI Write PIO	x0000_d030	x0000_d03f
10	HPI Chip-Select PIO	x0000_d020	x0000_d02f
11	HPI Reset PIO	x0000_d010	x0000_d01f
12	Pixel Render	x0000_0000	x0000_0fff
13	Dijkstra Core	N/A	N/A

The description of each components are as follows:

4.1 On-chip Memory RAM/ROM

On-chip memory for the NIOS II processor.

4.2 SDRAM Controller

Connects the SDRAM PLL 50MHz clock with the SDRAM controls through a conduit, i.e., the read, write, chip-select, address, and data signal of the SDRAM.

4.3 SDRAM PLL

Generates a clock used for SDRAM. The clock is skewed by -3 ns, Since there will be delay when communicating from on-chip to the SDRAM. This clock is set as an exported wire for use in the SV modules interface.

4.4 System ID Peripheral

Map components to designated System IDs, as a fail-safe to check for any ID inconsistencies that can arise due to improper handling (e.g., not generating modified Platform Designer design).

4.5 JTAG UART

Serves as to provide an interface between the Eclipse's console and the C program running during debugging.

4.6 Keycode PIO

This is the data read from keyboard presses. Each keys on the keyboard has their own values, and will be stored here in every key press.

4.7 HPI Read/Write/Chip-Select/Reset PIO

Read, Write, Chip-Select, and Reset signal for the HPI I/O interface, controlled by the NIOS II softcore processor.

4.8 Pixel Render

Interfaces the VGA, external input (keyboard and mouse), the control unit, and data from the master module (Dijkstra Core) to provide screen drawing support and quick response to user/computer input.

4.9 Dijkstra Core

Interfaces with the control unit, external input (starting and endpoint that is specified by mouse clicks), and the Pixel Render module after processing.

5 Design Statistics and Discussions

The following is the table containing the design statistics:

Parameters	Values
LUT	3631
DSP	8
Memory (BRAM)	350208
Flip-Flop	2532
Frequency	69.65 MHz
Static Power	102.27 mW
Dynamic Power	48.8 mW
Total Power	229.15 mW

In this design, we mostly use BRAM as the dynamic storage instead of instantiating registers. This added complexity in both reading and writing, and also value resetting but it brings the advantage of lower compilation time. Further, having all the memory as BRAM allows it to be more integrated when moving `DijkstraCore.sv` and `PixelRender.sv` into the SoC. With the low compilation time, it is easy to do debug, and hence accelerate the development process.

In this design, we also tried to minimize the usage of DSP blocks, which is mostly achieved due to our design utilizing mostly powers of 2. For example, addressing the tiles on screen is easy since the width is 512 pixels, and the tiles are 8 pixels wide, hence given a coordinate (x, y), the operation $x + y \cdot 64$ can be rewritten as `x+y << 6`. This minimizes the number of multiplications, which helps reduce the number of DSP blocks and also decrease latency time between operations.

6 Conclusion

Throughout our design, we experienced numerous bugs, such as screen distortion, which we fixed through better timing handling (and using the VGA Clock for R, G, and B output), seemingly random path that is drawn, which we fixed by making sure that the timings are correct (we added two state for Bound checking of each neighbors to account for the 1 cycle latency of reads of a BRAM), and path data not being drawn on the screen, which we fix by upgrading the OCM for the tiles data (`PixelOCM`) to be a true dual-port RAM, and create a selector using a MUX to ensure that writing operations are given as priority to `DijkstraCore` instead of user input during the path-finding phase.

To summarize, we were able to implement a paint program, where user can draw and select up to 64 colors on a 512 by 480 screen, with each drawn tile being 8 by 8 pixels in width. The user can then run a path-finding module to find the shortest path between two nodes given any obstacles that the user place, and can accurately determine if the path is possible to take in the first place or not.