Design Report: Paint Program with Dijkstra (BFS) Shortest Path Implementation

Loigen Sodian, Isaac Koo Hern En

May 2024

1 Introduction

The design being presented is a paint program, where users can draw on the screen using up to 64 different colors on a 512 by 480 screen, being able to draw an 8 by 8 tile (which means in total the user can draw 3840 tiles on the screen). Additionally, the user can also draw a starting and ending point on the screen, for which when run, will find a shortest path between the two points and plot it out on the screen. Now, since the distance between each pixel is the same, then the Dijkstra algorithm implementation will essentially become a BFS implementation.

After the shortest path has been found (or cannot be found due to no valid path), the user can optionally save the result in the SRAM, for which they can extract the data using the DE2-115 control panel to their computer.

2 Block Diagram

Figure 1 shows the block diagram:

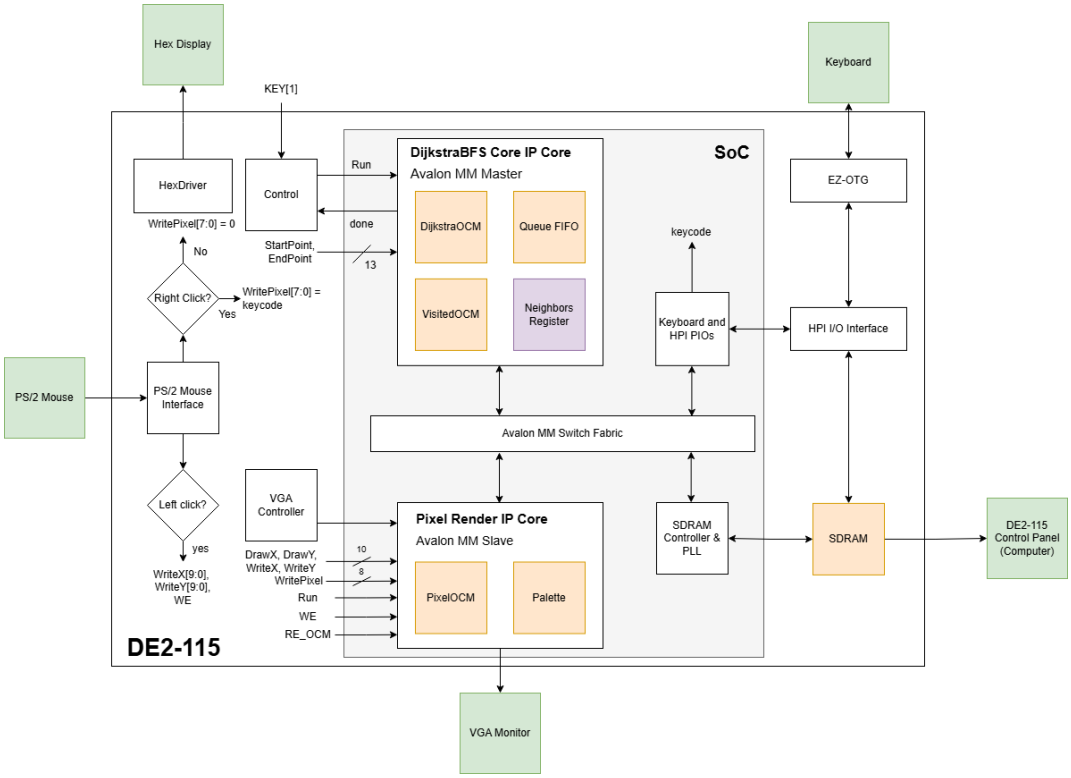

Figure 1: Block Diagram. Green represents external connections, orange indicate RAM or ROMs, while purple indicate registers

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:

2.1 On-chip Memory RAM/ROM

On-chip memory for the NIOS II processor.

2.2 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.

2.3 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.

2.4 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).

2.5 JTAG UART

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

2.6 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.

2.7 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.

2.8 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.

2.9 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.

3 Three Main Modules

The three main modules of this design is the `DijkstraCore`, `PixelRender`, and `Control`.

3.1 `Control` module

This module serves to control signals used for drawing and path prediction. It has 5 main states, DRAW, VERIFY, PROCESS, FINISH, RESET. 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 Start (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 FINISH, where the user can see the path being drawn. After this state, the user can opt to save the result into the SRAM (for reading through the control panel), or RESET, where it will reset all the RAM memory to zero (i.e., start with a blank canvas again).

3.2 `PixelRender` module

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

3.3 `DijkstraCore` module

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.

4 C Code

The C code used will be purely just to interface the keyboard with the whole module.