MAZE

A SIMPLE GPU WITH EMBEDDED VIDEO GAME

Designed by:

Joseph Liang	cjl2104@columbia.edu
Joe Zhang	xz2025@columbia.edu
Wei-Chung Hsu	wh2138@cs.columbia.edu
David Lau	dsl2012@columbia.edu

1. INTRODUCTION

Our goal is to design a simple GPU processor that could handle 2D line drawing function to implement a MAZE game. The user will be able to control a graphical object to traverse through a generated maze using the keyboard.

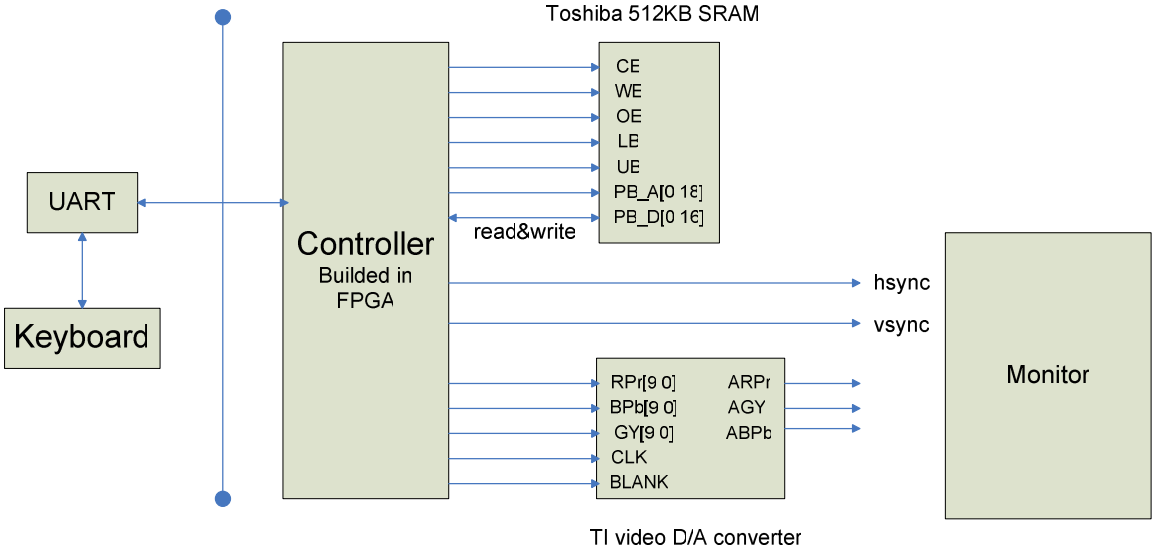
There are 3 parts of our project, (1) hardware functions, (2) firmware/driver/API functions, and (3) software game logic. For our hardware, we will utilize the UART to establish connection between the keyboard and the FPGA. Within the FPGA we will implement the Brensenham's line drawing algorithm in VHDL, which will be translated to an API function and called by software. We will then code up a game engine with C to run the logic, which will produce signals to communicate with the external Toshiba 512KB SRAM and the Texas Instrument Video D/A converter to display graphics.

2. BACKGROUND

API and Firmware, such as OpenGL [1] and DirectX [2], offers some functions that could direct use the hardware functions. The reason of API and Firmware is because it always writes some registers and memory space in order to trigger the hardware function, and it is inconvenient for software programmer directly writing these registers. In addition, some API may combine several hardware functions to create another powerful hardware function, for example, by calling drawing line function three times, it could get the simple drawing triangle function.

There are several similar projects done in previous semesters, and we consulted their design idea. The "BattleSnake" [3] and "Video Game" [4] projects give us pretty good example in character graphics rendering, and we may import some of their source code. The "Scorched Earth XESS" [5] demonstrates an excellent hardware design and utilization that they efficiently use the onboard Toshiba 256K x 16 bit SRAM.

3. OVERALL ARCHITECTURE (MODIFY THIS)



4. COMPONENTS

4.1 Toshiba 256K x 16 bit SRAM

We implement our video memory into Toshiba 512KB SRAM. For each color frame with 640x480 resolution, it takes 300KB, and this would limit us to implement a double buffer swapping function. Therefore, we will support one frame rendering for now. If we could figure out how to use the onboard Samsung 8Mx16 SDRAM, we maybe support swapping function in the future. The data buffer has read&write feature, therefore, it could load the game graphics data in the game loading time. As what the Lab6 did, we will directly import the Lab6 source code.

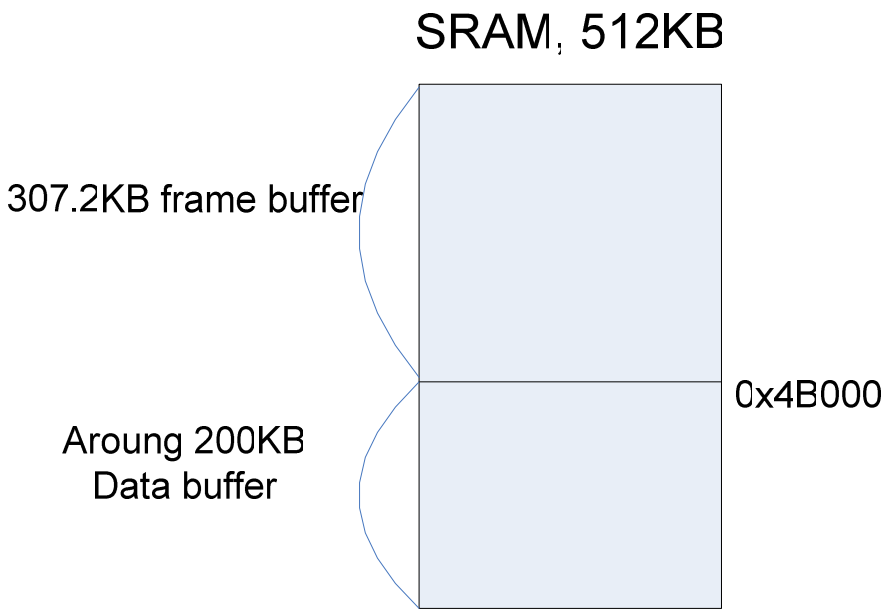


Figure 1: This is our SRAM usage

4.2 Texas Instruments video D/A converter

This chip is used to video out, as the lecture and lab5 source code did. We will directly import lecture and lab source code to use this peripheral.

5. HARDWARE ALGORITHMS

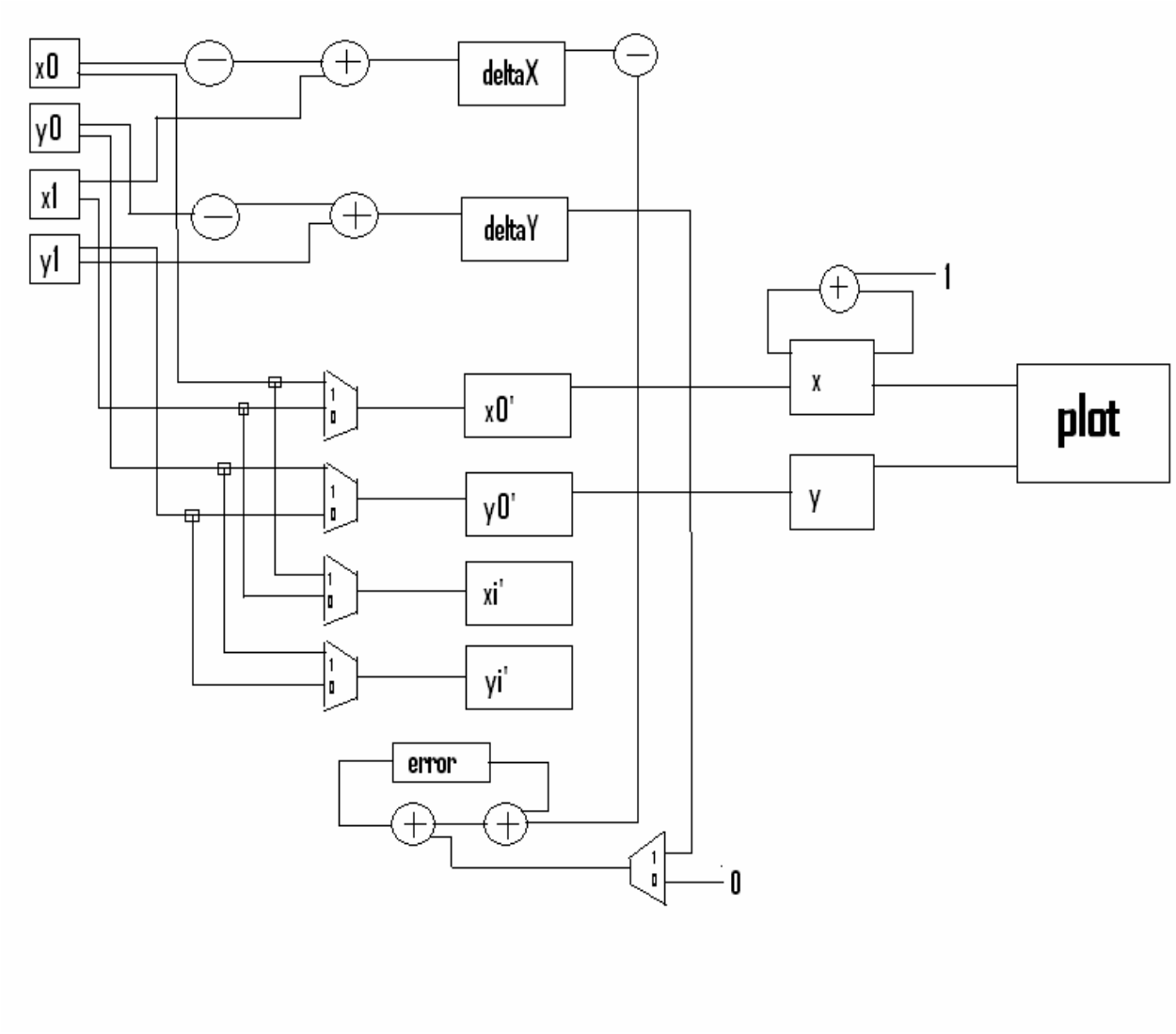
3.1 Bresenham's Line Algorithm

We want to implement a hardware drawing line function. The following is Bresenham's Line Algorithm pseudo code [6]:

```
function line(x0, x1, y0, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    int error := 0
```

```
int y := y0
if y0 < y1 then ystep := 1 else ystep := -1
for x from x0 to x1
    if steep then plot(y,x) else plot(x,y)
    error := error + deltay
    if 2×error ≥ deltax
        y := y + ystep
        error := error - deltax
```

The following block diagram is for the Bersenham's Algorithm:



6. SOFTWARE ALGORITHMS

6.1 Maze Generation

We will implement a random maze generator. Each time, the program will generate a random 50x50 maze. The way this is done is to start with all blocks in the 50x50 matrix in separate disjoint sets. Then, we randomly remove walls until all the blocks are in the same joint set. Each maze will have only one path from the starting block to the ending block.

6.1 User Interface

We allow the user to start at the starting point, and then he can use the arrows to control the direction in which he moves. A square represents the user. When the square reaches a pre-defined block, the program will detect a successful path out of the maze and it terminates.

7. FALLBACK PLAN & ALTERNATIVES

7.1 Fallback Plan

The backup plan is to simply generate a 50x50 maze, without letting the user to play.

7.1 Alternatives

We might also add color to our maze program using the Flood Fill Algorithm if time allows.

REFERENCES

[1] OpenGL: The Industry's Foundation for High Performance Graphics, http://www.opengl.org

[2] DirectX: provides a standard development platform for Windows-based PCs by enabling software developers to access specialized hardware features without having to write hardware-specific code, http://www.microsoft.com/windows/directx/default.aspx

[3] Ming-Ju Wu, Way-Cheng Sun. BattleSnake. In Embedded System Design Summer 2005.

[4] Dagna Harasim , Charles Finkel, David Soofian,, Ke Xu , Eric Li, Winston Chao. Video Game. In CSEE 4840 Embedded System Design 2004.

[5] Michael Sumulong, Jeremy Chou, Dennis Chua. Scorched Earf XESS. In CSEE 4840 Embedded System Design Spring 2005.

[6] Bresenham's line algorithm - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

[7] Foley, Van Dam, Feiner, Hughes, Phillips. Introduction to Computer Graphics.