EECS 150 Spring 2013 Checkpoint 4: Line Drawing Engine

Prof. John Wawrzynek
TAs: Vincent Lee, Shaoyi Cheng
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Revision 1, Due Wednesday April 17th, 2013 @ 2PM

1 Introduction

Hardware acceleration is a common technique used to complete tasks faster than possible in software. Additionally, because accelerators and the processor run in parallel, the processor is free to perform other tasks rather than spend computation on the accelerated task. In this checkpoint, you will implement hardware acelerated line drawing using Bresenham's algorithm.

2 Line Drawing Engine

We will be implementing the Bresenham's line drawing algorithm as presented in lecture.

Thus we recommend you review Bresenham's line drawing algorithm which can be found in:

http://inst.eecs.berkeley.edu/~cs150/sp12/agenda/lec/lec15-video.pdf

Implement this algorithm in `LineEngine.v`. The processor provides the line engine with the end-points of the line ($x0$,$y0$,$x1$,$y1$) and the color to draw. In a similar manner as the `FrameFiller` and `UART`, the line engine is controlled by a ready-valid interface via memory-mapped I/0. The interface to the `LineEngine` module is:

- `LE_color`, `LE_color_valid`: When in the idle state, the line engine should register `LE_color` when `LE_color_valid` is asserted.
- `LE_{x0, y0, y0, y1}_valid`, `LE_point`: When in the idle state, the line engine should register `LE_point` for the coordinate with its valid signal asserted.
- `LE_trigger`: When asserted, the line engine should begin drawing.
- `LE_ready`: Output indicating that the LineEngine is not currently drawing.
- DRAM FIFO Interface: Same as used in the `FrameFiller` module.

It is your job to correctly interface these signals from your processor.

3 Final I/O Memory Map

After adding the additional I/O for the memory mapped line engine and frame filler, the I/O memory map should now look like:

Table 1: I/O Memory Map

Address	Function	Access	Data Encoding
32'h80000000	UART transmitter control	Read	{31'b0, DataInReady}
32'h80000004	UART receiver control	Read	{31'b0, DataOutValid}
32'h80000008	UART transmitter data	Write	{24'b0, DataIn}
32'h8000000c	UART receiver data	Read	{24'b0, DataOut}
32'h80000010	Cycle counter	Read	Total number of cycles
32'h80000014	Stall counter	Read	Number of cycles stalled
32'h80000018	Reset counters to 0	Write	N/A
32'h8000001c	Filler Control	Read	{31'b0, FillerReady}
32'h80000020	Filler Color	Write	{8'b0, Color}
32'h80000024	Line Control	Read	{31'b0, LE_ready}
32'h80000028	Line Color	Write	{8'b0, Color}
32'h80000030	Line x0	Write	{22'b0, Point}
32'h80000034	Line y0	Write	{22'b0, Point}
32'h80000038	Line x1	Write	{22'b0, Point}
32'h8000003c	Line y1	Write	{22'b0, Point}
32'h80000040	Triggering Line x0	Write	{22'b0, Point}
32'h80000044	Triggering Line y0	Write	{22'b0, Point}
32'h80000048	Triggering Line x1	Write	{22'b0, Point}
32'h8000004c	Triggering Line y1	Write	{22'b0, Point}

4 New BIOS Command

The bios now has a command that will configure and trigger the hardware line engine. Remember to rebuild the bios ROM after making the changes. The syntax for new command is:

```
hwline <color> <x0> <y0> <x1> <y1>
```

5 Testing

The skeleton files include a line engine testbench that prints the points generated by your line engine. You can then compare the output to the C implementation of the line drawing algorithm in `software/le/`.

In order to compile the software test, simply use `gcc` and generate the object file. The binary takes four arguments:

```
./le.o <x0> <y0> <x1> <y1>
```

Where `(x0, y0)` denotes the coordinates of the origin of the line and `(x1, y1)` denotes the end coordinate of the line.

You should then run the hardware test harness which will generate the points that your line engine outputs and `diff` the two outputs. If they match, then you are good to go.

6 Memory Architecture Overview

Though the staff have provided the memory architecture for this stage of the project, it is useful to understand the high-level structure. Fig. 1 shows a block digram of the final processor and memory organization:

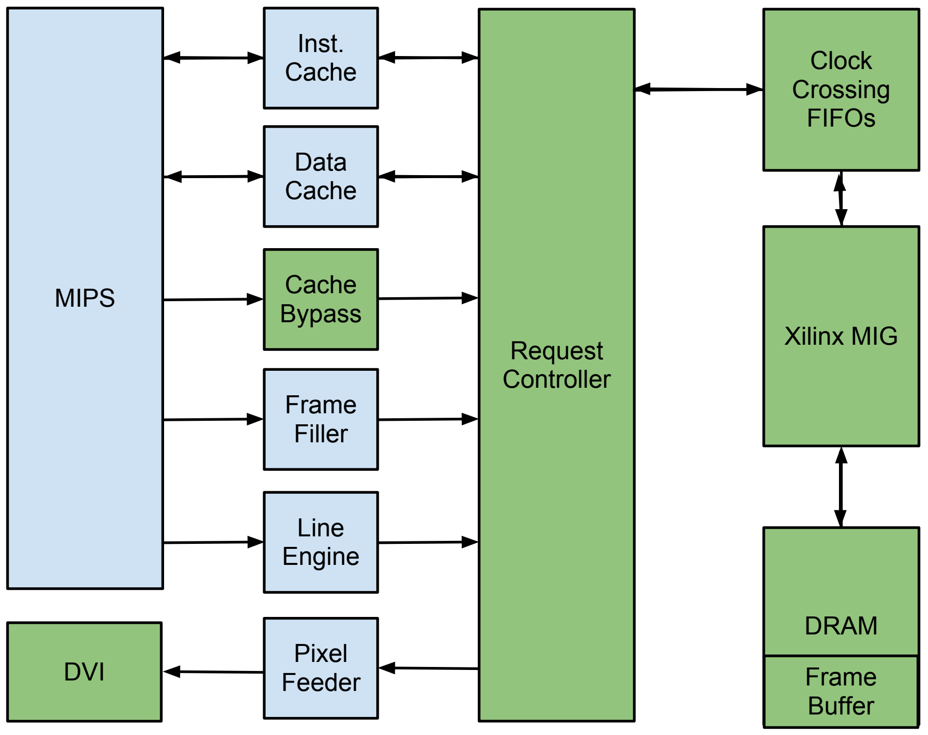

Figure 1: System Structure

Note that the instantiations of all of the modules accessing memory after inside Memory150.v which you should not have to modify.

7 Checkoff

Checkoff will consist of demonstrating hardware accelerated line drawing and comparing it against the speed of the software line drawing. This checkpoint is due at 2 PM on Wednesday, April 17, 2012. This is the final required checkpoint of the project.

8 Post Checkpoint

After finishing this checkpoint you are free to add any feature that you would like to your processor. You will be required to perform a demonstration of your final system so we highly recommend that you add features to your processor that will enahance your demonstration. In the past this has ranged from circle drawing hardware accelerators, graphical command prompts, animations, and simple games. This is the one part of the project where you can be creative and have fun (if your up for it).

9 How to Survive This Checkpoint

This is the easiest checkpoint in the project. Basically all you need to do is manually cross compile the C implementation of the line drawing algorithm into Verilog. We highly recommend that you use the testing framework that is provided to you. Some of the commonly made mistakes in this checkpoint usually stem from storing to the DRAM incorrectly. If you're lines look like they've been reflected about the y-axis, and weren't what you intended, you probably stored to DRAM the wrong way. Other than that, this checkpoint is fairly striaght forward.