EECS 150 Spring 2012 Checkpoint 5: Line Drawing Engine

Prof. John Wawrzynek
TAs: James Parker, Daiwei Li, Shaoyi Cheng
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Revision 1

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 accelerated line drawing using Bresenham's algorithm.

2 Line Drawing Engine

Review Bresenham's line drawing algorithm as presented in lecture:

`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 endpoints 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.

3 Final I/O Memory Map

The I/O memory map is now:

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}

The "triggering" commands should both write the point into the register and trigger the line engine to start drawing.

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/line-engine/`.

6 Checkoff

Checkoff will consist of demonstrating hardware accelerated line drawing. This checkpoint is due by 5 PM on Friday, April 20, 2012.

A Memory Architecture Overview

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

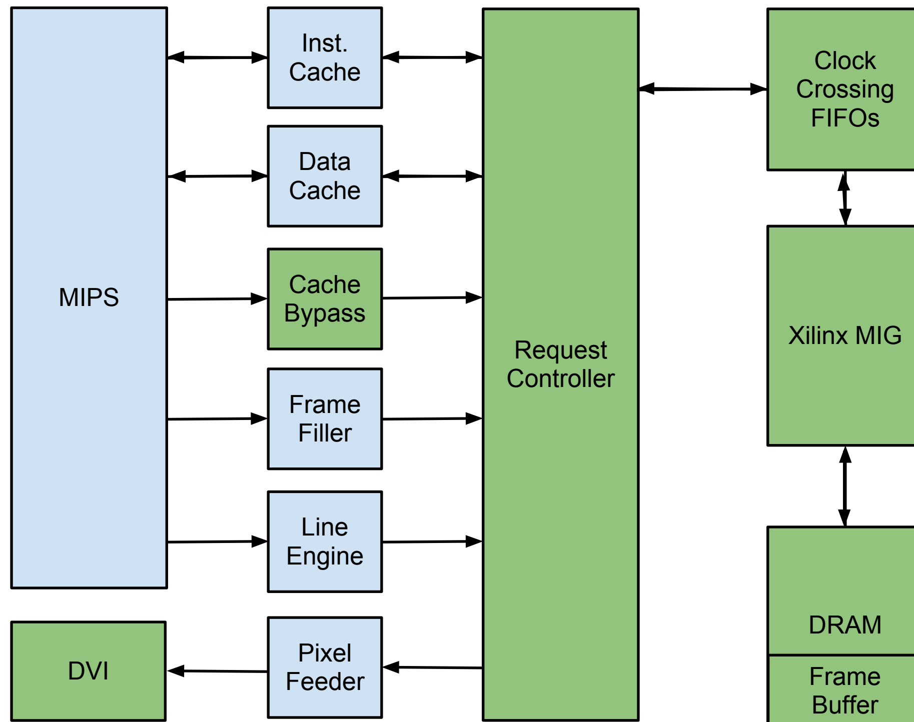

Figure 1: System Structure

The green modules have been provided in the skeleton files. As shown in the figure, there is only one way to access the DRAM: via the clock-crossing FIFOs that bring commands and data into the 200 MHz clock domain of the DRAM controller. With the addition of caches, DVI, and graphics accelerators, there are several blocks operating in parallel that access DRAM. This presents a challenge: only one command may be issued per cycle, and data read from DRAM needs to be directed to the block that requested it. Finally, because of the bandwidth demands of graphics, reads must be interleaved (i.e. after one read, another module must be able to read before the first reader's data is returned).

The RequestController module (typically called a memory arbiter) sits in front of the clock-crossing FIFOs and is responsible for granting access to the FIFOs and directing read data to the appropriate destination. Each module that accesses the FIFOs has a priority[1], and the highest

priority module attempting to issue a DRAM command is allowed to write to the clock-crossing FIFOs. This is accomplished by asserting the FIFO full signals to the blocks not granted access.

The read logic is simplified by the fact that there are only three modules that read from DRAM: the two caches and the pixel feeder. Furthermore, because the caches block on reads, each cache will have at most one read command in-flight at any time. Given this, interleaving the reads becomes fairly straightforward: the controller maintains a count[2]of read commands issued and reads serviced. When one of the caches issues a read, the number of the request is registered. That number (and a valid bit) is then used to direct the data to the cache when the serviced reads counter equals the request number. Otherwise, the data must have been requested by the pixel feeder. Directing the data is accomplished by asserting the read data valid signal only to the block that issued the request.

[2]The order is Instruction cache, data cache, pixel feeder, frame filler, line engine and finally the cache bypass.
[2]Using 10 bits, to ensure every read in-flight has a unique ID under worst-case scenario.