Checkpoint 5: Graphics Engine and Interface
CS 150, UC Berkeley, Fall 2011

0 Introduction and Motivation

Now that we’ve implemented an interface to DDR2 and caches for our CPU, we can run arbitrarily (almost) large programs and also have a large memory space for use. We will make use of a portion of this memory space for storing pixel data that we will then draw to a monitor attached to the FPGA board via a DVI->VGA adapter. We will also implement simple graphics acceleration in the form of a line drawing engine.

An introduction to line drawing:
http://inst.eecs.berkeley.edu/~cs150/fa10/Lab/CP3/LineDrawing.pdf
Also useful:
http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Fundamentally, we will be loading the pixel information into DDR2 memory by using MIPS instructions to trigger two memory mapped devices: a frame filler and a line engine, which will be responsible for setting specific addresses in memory. The portion of memory that contains the pixel information will be called the frame buffer. Simultaneously, we will have a “pixel feeder” that will grab pixels from the frame buffer in DDR2 memory and pass it to the DVI module, which is responsible for interacting with the chip that sends the video signal to the monitor. In order to deliver pixels at the correct rate to the DVI video module, the pixel feeder will operate in parallel with the MIPS processor, the frame filler, and the line engine.

We will again be using a memory-mapped architecture in order to communicate with the frame filler and the line drawing engine. For this checkpoint, the memory architecture becomes more complicated, as there are several blocks requiring access to the DDR2 memory. The request controller must process these simultaneous requests, decide who has priority, and interleave requests to the DDR2 to maximize performance (essentially, it must keep track of who the readers are to know who to return read data to after it comes back from DDR2, see RequestController.v for details).

Finally, the pixel feeder must read out the frame at a high data rate. Due to the high latency of the DDR2 access, it buffers the pixel stream using a FIFO. The pixel feeder must try to keep the FIFO full with pixel data from the frame buffer. At the same time, the DVI module will grab data out of the FIFO when as a new frame is being drawn.
1 DVI

The DVI module wraps up all of the low level details of producing video and provides a simple ready/valid interface for passing a 24-bit pixel (8 bits for each of red green and blue) at a time to the video hardware. Pixels must be supplied on the cycle that the DVI module is ready for them or there may be video errors. The module has been instantiated for you in the skeleton and should be fairly self-explanatory.

2 Pixel Feeder

In order to actually have pixels to supply to the DVI module you need to store them in memory. The only place you will have enough memory to do this is in the DRAM. The latency for doing a DRAM read is too long to read pixels out one at a time, so you will need a FIFO to buffer pixel data coming out of the DRAM going in to the DVI module and a pixel feeder to keep the FIFO full. We will provide a pixel feeder, however, you are encouraged to explore how this module works.

There are a variety of memory layouts you could choose for the pixel feeder, but we will use one word per pixel (wasting 8 bits) and store pixels in row-major order. Also, consider that your line engine will need to write to a memory address using only an x and y coordinate. In order to make it easier on the line engine, we will choose to store each block of 8 pixels at \{6'b0, 6'b1, y, x[9:3], 2'b0\}, where x and y are 10 bit values. Note that we are cutting off the last 3 bits of x to align pixels (32-bits) to a DDR2 burst (256-bits). In other words, every memory block contains 8 horizontally adjacent pixels; as discussed later, you will therefore need to use the memory mask to write to an individual pixel in the block.
3 Line Engine

Your line engine should implement Bresenham's line drawing algorithm. You will find an implementation of the algorithm written in C in `hardware/sim/compare/le.c`. The CPU will provide coordinates for the end points of the line and a color to draw the line in. The memory-map interface for the line engine is shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEC0</td>
<td>0x80000030</td>
<td>Read</td>
<td>1 if the line engine is ready to draw a new line, 0 otherwise.</td>
</tr>
<tr>
<td>LECr0</td>
<td>0x80000034</td>
<td>Write</td>
<td>The color of the line.</td>
</tr>
<tr>
<td>LEX0</td>
<td>0x80000040</td>
<td>Write</td>
<td>X coordinate of the first end point.</td>
</tr>
<tr>
<td>LEY0</td>
<td>0x80000044</td>
<td>Write</td>
<td>Y coordinate of the first end point.</td>
</tr>
<tr>
<td>LEX1</td>
<td>0x80000048</td>
<td>Write</td>
<td>X coordinate of the second end point.</td>
</tr>
<tr>
<td>LEY1</td>
<td>0x8000004c</td>
<td>Write</td>
<td>Y coordinate of the second end point.</td>
</tr>
<tr>
<td>LEX0Trigger</td>
<td>0x80000050</td>
<td>Write</td>
<td>Triggering version of LEX0.</td>
</tr>
<tr>
<td>LEY0Trigger</td>
<td>0x80000054</td>
<td>Write</td>
<td>Triggering version of LEY0.</td>
</tr>
<tr>
<td>LEX1Trigger</td>
<td>0x80000058</td>
<td>Write</td>
<td>Triggering version of LEX1.</td>
</tr>
<tr>
<td>LEY1Trigger</td>
<td>0x8000005c</td>
<td>Write</td>
<td>Triggering version of LEY1.</td>
</tr>
</tbody>
</table>

Most of these should be fairly self explanatory. The triggering versions of the coordinate ports both update the coordinate and cause the line engine to begin drawing a new line. Note that in the skeleton of the line engine provided, there is only one trigger input. This means that you should set this input high when writing to any of the four trigger addresses.

Your line engine will need to set pixel data in DDR2, thus it has the necessary DDR2 FIFO connections to write data to DDR2. Note that since we have provided the pixel feeder for you, you will need to use the same addresses for your pixels that are used by the pixel feeder. Specifically, you will want to set \( ax_{addr_{din}} \) to \( \{6'b0, 6'b1, y, x[9:3], 2'b0\} \), where \( x \) and \( y \) represent the coordinates of the pixel you wish to write to (they are both 10 bit values), and then set \( wdf_{mask_{din}} \) appropriately (be careful with this, since each write is split into 2 128-bit chunks) depending on the pixel you want to write. Note that since DDR2 expects a 64-bit aligned address, you should put the color information in the correct part of \( wdf_{din} \). You will want to write pixels into DDR2 in a big endian\(^*\) order, which means that if \( x_{[2:0]} = 3'b0 \), you will want to set \( wdf_{mask_{din}} \) of your first write to DDR2 to be \( 0x0FFF \). The simplest implementation will write one pixel (32 bits) per 256-bit request to DDR2, while a more sophisticated implementation could try to buffer pixels to make larger requests.

\(^*\)This is because of the way we implemented the pixel feeder/frame buffer. We chose to simply use a FIFO to enqueue pixels going to the DVI module. As it happens, the FIFO expects pixels in a big endian order when you have asymmetric read/write widths. In our case, the write width is 128 bits (we read straight from \( rdf \) into the \( pixel_{fifo} \)), and the read width is 32 bit. This means if we wrote \( \{32'd1, 32'd2, 32'd3, 32'd4\} \) into the FIFO, the DVI module would read out \( \{32'd1\}, \{32'd2\}, \{32'd3\}, \{32'd4\} \).
3.1 Line Engine Testbench

The staff have provided a testbench (LineEngineTestbench.v) to assist in debugging your line engine implementation. This testbench displays the coordinates you are trying to write to whenever you set af_wr_en high. You can diff the transcript of this testbench with the output of drawing the same line using the C implementation of the line engine to check if your Verilog implementation is functionally correct by using the compare_lines script in the hardware/sim directory. Note that you will have to modify LineEngineTestbench if you decide to buffer pixel writes to DDR2.

3.2 Memory Mapped IO Testbench

For the optional procedures:

The staff have also provided a memory mapped IO testbench to ensure that you are asserting the correct signals when writing to your line engine and frame filler. You will need to change the mmio.do file to change the .mif file to match whatever you named your BIOS ROM. You can compare the output to the staff output (saved to hardware/sim/compare/MMIOTestbench.out).

4 Frame Filler

For the optional procedures:

In order to fill the background with a color, we could (incorrectly) try to write to the dCache at the memory address of the frame. However, the compiler can’t know what type of cache you have implemented (and this would be a terrible design methodology anyway), so we do not have a way to evict the new pixel information to the DDR2 and thus we don’t have control over when the new frame would show up.

Instead, we will add a frame filler interface (ColorFiller.v). The memory map is:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFControl</td>
<td>0x8000001C</td>
<td>Read</td>
<td>1 if the frame filler is ready (not busy), 0 otherwise.</td>
</tr>
<tr>
<td>FFColor</td>
<td>0x80000018</td>
<td>Write</td>
<td>The color of the background.</td>
</tr>
</tbody>
</table>

This module is provided for you.

5 Software

For the optional procedures:

In order to actually make use of our frame filler and line engine, we will need to write software to take advantage of their capabilities. We now have 2 new commands in our BIOS:

- fill <00RRGGBB> (color is 32-bit hex)
- line <00RRGGBB> <X0> <Y0> <X1> <Y1> (coordinates are decimal)

You may also wish to write your own software to draw creative things to the screen. To do so, make use of the line() and fill() functions defined in software/graphics.
6 FPGA Debugging

*For the optional procedures:*
If you can’t get your code to work on the FPGA and want to verify your line engine in simulation, you may want a way to read out data from the frame buffer. Note that this should only be a last resort, after fully debugging your implementation in simulation. However, to correctly implement a read path for the frame buffer requires modification of the RequestController in order to support this. Note that we have the same problem as trying to write the frame through the d-Cache: we don’t have control over the concurrency of the cache. Since reading out the frame by your MIPS is only used for verification and test purposes, we will simply accept the d-Cache concurrency limitations. This means that you can only read out a pixel through the cache once, unless you intentionally evict that cached pixel and then read it again. Again, you should ensure that simulations are correct before attempting to debug on the FPGA.

7 Required Procedure

1. Git
   a. Tag your CP4 design on git so you can easily roll back (and so we can find it for grading)
   b. Pull the latest skeleton for CP5 (its tagged)

2. Implement the line drawing engine
   a. A test bench is available for the line engine for testing it separate from your MIPS (LineEngineTestbench.v). You can verify your line engine as described in the previous sections.

8 Optional Procedure

1. Integration of new memory request controller, pixel feeder, line engine and frame filler
   a. We will announce on Piazza when these modules are ready. At that time, pull the skeleton updates and resolve any merge issues. This should be really easy, and afterwards, you will be ready to start drawing!

2. Add the memory mapping hardware
   a. Modify your MIPS to support writes to the Frame Filler and Line Engine (the memory map was given above)
   b. A test bench is available for verifying that your memory map is correct (MMIOTestbench.v). You will load a very simple program into your bios memory that calls the frame filler and line engines. You should see $display outputs verifying that you are interacting with the modules correctly and compare them against hardware/sim/compare/MMIOTestbench.out.

3. Create a demo
   a. See the Software section above for information on how to interact with the MMIO. You can write your own visualization program that draws to the screen, and load it from the bios. Feel free to integrate XUP buttons, other input devices, etc.
9 Checkoff

Checkpoint 5 Checkoffs (Friday, Dec. 2):
Demo a simulation showing that your line engine works correctly.

Optional Final Checkoff (Tuesday, Dec. 6):
Demonstrate that your design (MIPS+Memory+LineEngine+DVI) is capable of doing hardware accelerated line drawing by jumping to a drawing program from bios and drawing an image on the LCD screen. A simple demo is sufficient, but feel free to get creative!

We will make a signup list for checkoffs. There will be no checkoffs after Dec. 6.

Information on the final report and grading will be posted on the CS150 website.

Revision History:
Nov 30 - Added notes on endianness for the Line Engine