EECS 150 Spring 2013 Checkpoint 3: Video Interface

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 @ 2PM April 10th, 2013

1 Introduction

A framebuffer is now possible with the large amount of memory available to the processor. In this checkpoint, you will design a module to fill a frame with one color as well as complete missing portions of a module that transfers pixels from the framebuffer to the DVI. These tasks are designed to be an introduction to hardware acceleration: the modules operate in parallel with the MIPS CPU to complete tasks faster than possible in software.

2 Relevant Modules

- `PixelFeeder`: Recall that DRAM is high capacity but also high latency. This module sits between the DRAM and the DVI interface to buffer pixels from DRAM and transfer them to the DVI module. The FIFO and output logic have been provided; you will write Verilog to keep the FIFO filled with pixels.
- `FrameFiller`: This module accelerates filling the screen with one color. When the CPU stores a color to this module's address in the I/O memory map, the module should store the color to every location in the framebuffer. You will need to design this module.
- `CacheBypass`: In order to do software line (or other shape) drawing, the CPU needs to be able to write directly to the framebuffer. This module bypasses the cache and writes directly to DRAM (though the implementation is not efficient).

3 Framebuffer

The framebuffer is simply a region in memory where each word maps to a pixel on the display. The skeleton files have defined the framebuffer base as `0x10400000`. The addressing scheme is as follows:

```
address = {10'b0001_0000_01, y, x, 2'b0}
```

where x and y are ten bits. The output resolution is 800x600 and the base address corresponds to

the origin (top left) of the display. This scheme leaves unused portions of memory in the framebuffer but simplifies addressing. You will need to convert this address to the address space of the DRAM to complete `FrameFiller` and `PixelFeeder`.

Consult the lecture slides for review on framebuffers:
http://inst.eecs.berkeley.edu/~cs150/sp12/agenda/lec/lec15-video.pdf

4 First Step: Completing the PixelFeeder

The staff have provided a FIFO and output logic in `PixelFeeder.v`. The FIFO is asymmetric: the write port is 128 bits to match the width of one cycle of data from DRAM, and the output is 32 bits, the size of one pixel in memory (though the top 8 bits are unused).

Your task is to design an FSM that keeps the FIFO as full as possible. The challenge is the latency of the DRAM - you must ensure there is room on the FIFO when the requests are serviced. You'll likely need logic to count the total number of pixels requested and in the FIFO.

The latency leads to one other slight complication in the output logic: After reset, the DVI module begins requesting pixels before it is possible to read from DDR2. To alleviate this, the first frame after reset is dropped. This is done in the skeleton code with a counter called `ignore_count`. Make sure to include this in your counting logic as you complete the module.

Once the module is working, you should see a stable image on the display.

5 Second Step: Accelerated Filling

Complete the `FrameFiller` module. Similar to the UART, this module uses a ready/valid interface via memory-mapped I/O to communicate with the CPU. When the CPU stores a word to the module, the FrameFiller should write the word to every location in the framebuffer.
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

6 Third Step: Cache Bypass

The cache bypass module has been provided for you and instantiated in DDR2. However, like the caches, you'll need to setup the inputs based on the address. The data cache should be bypassed on stores with bit 30 of the address set high. The memory partition is now:

Table 2: Updated Memory Address Partitions

Address[31:28]	**Address Type**	**Device**	**Access**	**Notes**
4'b00x1	Data	Data Cache	Read/Write	
4'b0001	PC	Instruction Cache	Read-only	
4'b001x	Data	Instruction Cache	Write-Only	Only if PC[30]
4'b0100	PC	BIOS memory	Read-only	
4'b0100	Data	BIOS memory	Read-only	
4'b0100	Data	Cache Bypass	Write-only	
4'b1000	Data	I/O	Read/Write	

7 New BIOS Commands

The bios has been augmented with a commands for software line drawing and filling the frame. Remember to rebuild the bios ROM after making the changes. The syntax for new commands is:

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

fill <color>
```

8 DDR2 Request FIFO Interface

The XUPV5 development board has a 256MB DDR2 SODIMM (small outline dual inline memory module) mounted on the underside. The interface to this module is provided through Xilinx's MIG (memory interface generator), which in turn is connected via clock-crossing FIFOs to the memory arbiter. The arbiter sits between the caches and the MIG and provides the illusion that each cache has exclusive access to the FIFOs.

Inputs to the cache from the arbiter:

- `rdf_dout`: 128 bits of data out from the read data FIFO.
- `rdf_valid`: Indicates the data from the read data FIFO is valid.
- `af_full`: Indicates the address FIFO is full. You can think of this as a ready signal.
- `wdf_full`: Indicates the write data FIFO is full. You can think of this as a ready signal.

Outputs from the cache to the arbiter:

- `rdf_rd_en`: Read-enable signal for the read data FIFO. You can think of this a ready signal.

- `af_cmd_din`: 3-bit command to the DDR2 controller.
- `af_addr_din`: Address (in the DDR2 domain).
- `af_wr_en`: Write-enable signal for the address FIFO (which also writes the command). You can think of this as a valid signal.
- `wdf_din`: 128 bits of data to the write data FIFO.
- `wdf_mask_din`: 16-bit active-low byte write mask.
- `wdf_wr_en`: Write-enable signal for the write data fifo (includes the mask). You can think of this as a valid signal.

The memory on the board is configured for a burst length of 4 and each address maps to 64 bits of data. DDR stands for 'Double Data Rate', which means that data is transferred on both edges of the controller's clock. The controller therefore has a port width of 128 bits that is connected to the clock crossing FIFOs. Finally, to exploit the efficiency of reading in bursts, a natural block size for the cache is 256 bits.

For the cache, this means the following steps need to be performed for a write:

1. Supply a 31-bit address to `af_addr_din`, of which the low 25-bits matter, while the upper 6 should be zero.
2. Set `af_cmd_din` to 3'b000.
3. Supply 128-bits worth of data to `wdf_din`.
4. Supply 16-bits worth of byte mask to `wdf_mask_din`.
5. Assert `wdf_wr_en` and `af_wr_en` then wait for a handshake with `!af_full && !wdf_full`.
6. Supply the next 128 bits of data and assert `wdf_wr_en` and wait for a handshake with `!wdf_full`.

Then, to read:

1. Supply a 31-bit address to `af_addr_din`, of which the low 25-bits matter, while the upper 6 should be zero.
2. Set `af_cmd_din` to 3'b001.
3. Assert `af_wr_en` when `!af_full`.
4. Assert `rdf_rd_en` to indicate waiting for data.
5. Wait for `rdf_data_valid` to be asserted and store the first half of the block.
6. Wait for `rdf_data_valid` to be asserted again and store the second half of the block, and set `rdf_rd_en` low again.

9 Checkoff

This checkpoint is due 2 PM, Tuesday, April 10. Checkoff will consist of filling the screen with a color and drawing a software line from the origin to the center of the screen. Note you do not need to finish the line engine (hardware accelerated version) for this checkpoint. It will be complete next checkpoint and should not cripple any functionality for this checkpoint.

10 How to Survive This Checkpoint

This checkpoint should be more straightforward than previous checkpoints. We recommend that you first get the pixel feeder to work. You can tell the pixel feeder works, if you see the RGB rainbow on the screen (DRAM garbage). Unfortunately it's hard to simulate a display so testing this module is either going to be chipscope or trial and error so don't guess at it. There are some subtleties to get the pixel feeder algorithm right. Once you get your display to read out of DRAM correctly, we then recommend that you do the framefiller and cache bypass since you can directly test these work once you have the display working correctly. Note that if you make impact to the board, it does not reset the contents of the DRAM. Thus if you are unsure if someone else trashed the DRAM before you did, you should power off the FPGA and power it back on before testing your design. Powering off the FPGA is the only way to completely reset the DRAM to garbage so that it doesn't mask any bugs or make it harder to debug.