EECS150: Fall 2010 Project Checkpoint 2, SRAM Framebuffer & Video

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

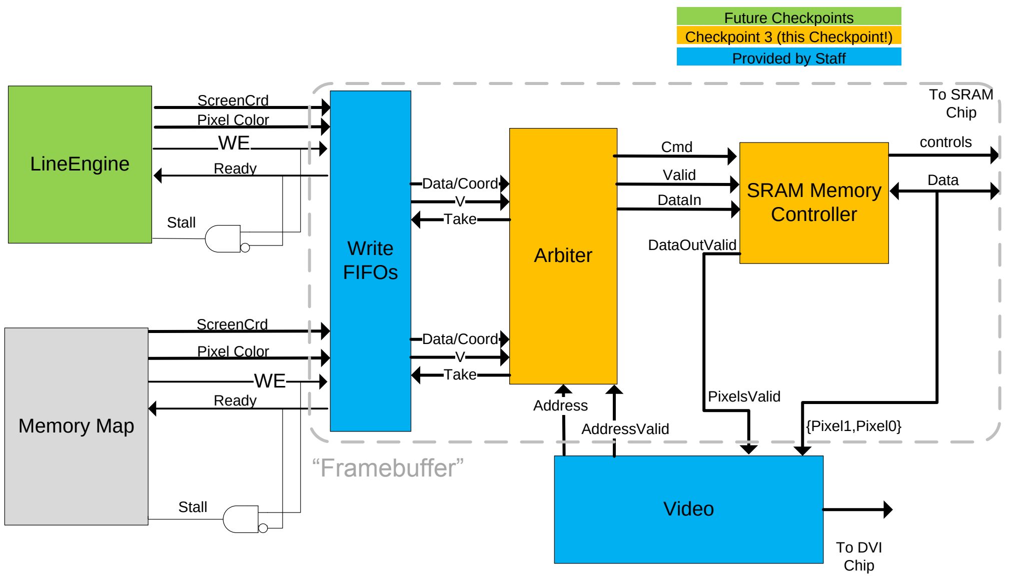
Revision E

1 Time Table

ASSIGNED	Friday, October 29th
DUE	Week 12: November 9th – 12th

2 Motivation

The next phase of the project will involve building a video subsystem. This consists of a Framebuffer, which will require an arbiter, memory controller to communicate with off-chip SRAM, a video interface, and a line drawing acceleration engine. This checkpoint will involve building some and integrating all of those components except the Line Engine.



3 SRAM

The XUPv5 board has a 256Kx36 synchronous ZBT[1] SRAM chip. It supports a data width of 32 bits and 4 parity bits. Although parity bits can be used for data, you are not required to make use of them. The

[1]ZBT: Zero bus turnaround, which means that the data bus require does not require any cycles between read and write operations

SRAM exposes a standard single-port memory interface with a single data bus, which means that reads and writes share the same bus for data. The SRAM has burst read/write capability, but you will not need to use bursts. You will be responsible for implementing an SRAM memory controller, which is the interface between your design and the SRAM device. `SRAM` documentation is on the Documents page of the website. The controller interface with the SRAM chip is already fixed. *The controller interface with your design is up to you*, but you might include ports for Address, Read/Write Request, ValidRequest, RequestData (writing), ByteMask, ResponseValid (reading).

The SRAM device is ZBT, so a Read command can be followed by a Write command the following cycle and vice versa. Also, it is possible to do back-to-back writes and back-to-back reads. In short, on every cycle you can issue a read or a write to the SRAM.

As you will find from the documentation, the SRAM is a synchronous read/synchronous write device, and the latency to complete an access is 2 cycles. Accesses can be pipelined, but keep the latency in mind when designing this checkpoint.

Some signals going to the SRAM device are "active-low", meaning they are interpreted as being asserted when their value is 0, rather than 1. These signals are distiguished in the documentation using a bar over the name.

In the SRAM documentation, you will be most interested in reading pages 1-2 (intro), 8 (pin definitions), 9&11 (truth tables), 18-20 (timing diagrams), and maybe 21-22 (extra timing diagrams). It might also be helpful to read the verilog simulation module supplied by the manufacturer, and available in the project docs.

4 Framebuffer

The **framebuffer's** responsibility is to store an entire frame of video in the FPGA fabric so that the video subsystem always has data availible. This is a common problem with video interfaces: if the video component does not get exactly the data that it needs *when it needs it*, the image will appear broken or even not show up at all. Having a full frame worth of data ready "on demand" ensures that the rate at which video data is required by the video subsystem is always satisfied. The conceptual Framebuffer in this project is outlined in the diagram above.

4.1 Implementation with SRAM

The SRAM has capacity for 256K 36-bit words. An 800x600 display contains 480,000 pixels, and our framebuffer needs one entry for every pixel. So how many bits per pixel? $256K * 36bits/480000pixels = 19bits/pixel$. In principle you could use the full 19 bits per pixel, but you can imagine that memory references would get ugly, with pixels stored across different memory words, etc. A good choice would be to represent the color of a pixel using 16 or 18 bits, which means two pixels per SRAM data word. Every access to SRAM could read or write two pixels, but only if these pixels are a sequential even-odd pair.

The video interface expects pixels to be stored sequentially in the SRAM, at location 800Y+X. For the CPU and Line Engine it is more convenient to write to locations in pixel coordinates, (Y,X). You will need to implement the conversion from pixel coordinates to sequential addresses (see section on Address Generator below).

4.2 CPU Interaction

The CPU–and eventually the Line Engine–must be able to write one color at a time to a pixel location in the framebuffer. From the perspective of the CPU, the framebuffer will be another memory-mapped IO device. The mapping is shown in Table 1. The CPU writes to pixel coordinates (Y,X), so addresses

Table 1 Framebuffer memory map.

`0x80000000-0x803ffffc`	W	Framebuffer

are of the form {0x80, 00, Y 10 bits, X 10 bits, 00}. Giving a separate 10 bits to each coordinate is technically a waste of address space, but there are more than enough addresses available and the coordinate abstraction is convenient for software writing to the framebuffer.

Since the CPU writes only one 16-bit pixel at a time and the granularity of SRAM memory accesses is 32-bit words, you will need to make use of the ByteWriteEnable (BWa-BWd) signals going to the SRAM device. These specify for writes which of the four bytes in the word to write to and which to leave as is. What would the consequences be if these enable inputs did not exist?

The staff provides a module WriteFIFOs, which will provide a ready/valid interface for each the LineEngine and CPU Memory Map to enqueue data that is to be written to the Framebuffer. Notice on the figure that there are stall signals generated in the case where the queue is not ready for data. These will halt the LineEngine and CPU, respectively. To stall the CPU means that all stage registers will hold their value so as to pause execution. *Note: since it is a rare case for the CPU to need to stall (e.g. when it is writing to the Framebuffer at a high and sustained rate and fills up the queue), try to get the rest of the checkpoint working before implementing this.* When you build the LineEngine, on the other hand, you should build in stalls from the beginning. Below is the interface for WriteFIFOs. The DataOut ports are asynchronous read, synchronous dequeue (i.e. the next data becomes available after Take is high at the rising edge of the read clock).

Here is the port interface to the Video module.

Table 2 WriteFIFOs ports.

Name	*Width*	*Description*
`CPUDataOut`	16	Pixel color to write, coming out of FIFO.
`CPUCoordOut`	20	Pixel location to write, coming out of FIFO.
`CPUValidRequest`	1	There is valid data available to take.
`CPUTake`	1	The CPU write request was chosen and should be dequeued.
`CPUCoordIn`	20	Pixel coordinate address to be written to Framebuffer.
`CPUDataIn`	16	Pixel color to be written to Framebuffer.
CPUWriteEnable	1	"MemWrite" signal to Framebuffer.
CPUWriteReady	1	Ready to accept CPU writes (room in FIFO).
-	-	...indentical ports for Line Engine...

4.3 Arbiter

The SRAM memory is single-ported, yet we have three blocks using the framebuffer (CPU and Line Engine writing and video interface reading). The framebuffer will essentially look like a memory with 2 write ports and 1 read port (except with some Ready signals in places). Your design must have an "Arbiter" to choose between the memory users. A typical arbitration system involves each user providing a Valid signal to indicate whether it wants to make a "request" to the shared device (e.g. SRAM controller); the Arbiter chooses between the Valid requests and sends out the chosen request to the device, while also notifying the corresponding user that its request was "granted". Since you must design the Arbiter and SRAM Controller, their communication interface is up to you. The following constraints apply to the project:

1. The video interface read requests (indicated by AddressValid being high) have priority over CPU writes.

2. The CPU writes have priority over Line Engine writes.

Notes:

1. Since the SRAM is single ported and there are three users, using a FIFO for CPU writes to the Framebuffer has a benefit in performance, besides providing a clock crossing. Think of what the benefit is.

5 Video Interface

The Chrontel CH7301C chip on the XUPv5 board is a display controller device for transmitting data through the DVI output to a display device, like a LCD monitor. The staff will provide a video interface block (and an example of how to instantiate it within your design) that will communicate with this off-chip device. Your design will need to just communicate with this block to accomplish sending pixel data to a display device at the required rate. When AddressValid is high, the Arbiter should choose to issue a read request to SRAM Controller for Address. This Address is a sequential pixel address from 0 to 239999 (number of pairs of pixels for 800x600 resolution). When the 32-bit data (2 pixels) comes back from SRAM, the data should be transferred to the Video by asserting the PixelsValid signal.

The display specification we are using, SVGA Signal 800 x 600 @ 75 Hz, requires a pixel clock of approximately 49.5MHz.

The video interface takes in 24-bit color on the Video input. Since we choose 16 bits for our pixels, the 16 bits must be padded to 24 bits. The 24 bits are normally partitioned as 8 bit Red, 8 bit Blue, 8 bit Green. The partitioning and padding for 16 to 24 should be Red5, Green6, Blue5: {RRRRR000, GGGGGG00, BBBBB000}. Notice that the color bits should be the upper bits; otherwise, colors going to the video interface would all be too dark. The provided video interface implements this padding for you.

5.1 Address Generator

The video interface expects pixels in order from left to right and from top line to bottom line, in that order. Recall that we choose to store two 16-bit pixels in each SRAM 32-bit word. So, for maximum SRAM read bandwidth, pixels should be stored sequentially in this order. If we can read two pixels in one SRAM cycle and the video interface expects pixels at 50 MHz, then the read rate is 25 MHz. Since the SRAM Framebuffer will run at 100 MHz, SRAM reads take up only 1 in 4 available cycles (and none during blanking intervals). The Video interface will generate valid Addresses at the required rate.

If the SRAM stores pixels sequentially, but the CPU and Line Engine write to the Framebuffer using coordinate addresses[2], there must be address translation from coordinate to sequential address: $SeqAddress = 800*Y+X$. This translation is an interesting little block to optimize for performance/area because it involves a multiply-by-constant and an add. Note that just writing 800*Y in your verilog is NOT the right answer.

5.2 Interface

Here is the port interface to the Video module.

Table 3 Video ports.

Name	*Width*	*Description*
`Pixels`	32	Read data from the Framebuffer to the Video.
`PixelsValid`	1	High when data on `Pixels` is valid.
`Address`	18	Sequential pixel pair address of next pixel pair to read.
`AddressValid`	1	When high, indicates `Address` is valid and a read request should be issued for the address.

6 Requirements

You are responsible for implementing the following blocks:

1. Arbiter
2. SRAM memory controller

[2]Everyone's CPU memory map must use coordinate addresses so that any code the staff provides involving writes to the framebuffer will work. As for the Line Engine, you will see that coordinate addresses are a result of the drawing algorithm.

You must integrate the blocks into your design so that the CPU and video interface can interact with the Framebuffer as described in the previous sections. Refer to the diagram on the first page and interfaces.

You should develop some simple on-board test scenarios for your whole system up through this checkpoint. For example, write a simple program to fill the the Framebuffer with simple patterns (color bars are popular) and have it display correctly on a monitor hooked up to the DVI port on the board. Pick the pattern based on input on the UART0 CLI (Command Line Interface). If you had a pixel map of a particular font, you could write a program that would let you echo text from your CLI on to your own graphics screen.

7 Additional Information

The following sources will be helpful throughout this checkpoint:

1. `SRAM` documentation
2. Spring 2010 lectures 14, 15, 16 on framebuffer, video, SRAM.

Rev.	**Name**	**Date**	**Description**
E	Doupnik, Eastham, & Gaddam, Pister	10/27/2010	Updates.
D	Brandon Myers & Chris Fletcher	4/24/2010	Fixed an error that said back-to-back reads and back-to-back writes to the SRAM are not possible. Clarifications in top level figure.
C	Brandon Myers	4/16/2010	Removed Framebuffer-to-DVI, CPU-to-Arbiter communication from required blocks to implement.
B	Brandon Myers	4/11/2010	Added information about acceptable SRAM command sequences.
A	Brandon Myers & John Wawrzynek	4/1/2010	Designed new checkpoint. Concept adapted from Framebuffer checkpoint from Spring 2009.