Overview

In this project, you will use SDRAM to implement video frame buffers, fill them with the output of a graphics accelerator and implement a pixel feeder that pulls successive frames out of SDRAM and sends them to the DVI driver.

So far you have a UART interface that talks to an external PC with a screen and a keyboard. That PC handles all of the work associated with drawing characters on the screen. In this project you will generate a video signal to drive a separate video monitor. You’ll still have the UART connection to the PC screen and keyboard to use for command inputs and debugging outputs, but the video monitor will be where your game eventually gets played.

The video signal comes from a Chrontel DVI chip. The DVI chip generates an 800x600 pixel output frame 75 times per second. The Chrontel chip is driven by cs150-staff-supplied DVI driver module, which gets its pixels from a FIFO. Your job is to keep the FIFO full with a module called Pixel Feeder. The Pixel Feeder (PF) reads from frame buffers in SDRAM, and writes to the FIFO in the DVI driver. A frame buffer is a location in memory where each memory word corresponds to a particular pixel location on the screen. You will use two frame buffers in this project, known as double-buffering. This will allow you to draw lots of moving objects on the screen. Every pixel of every frame must be pushed by the Pixel Feeder. If you write software on the MIPS to fill a frame buffer with all zeros and have the pixel feeder push that frame to the DVI over and over, you will see a black screen. If you also fill a different frame buffer with all ones, that frame would be displayed as all white. Having the PF push those two buffers to the DVI driver on alternate frames would cause the video screen to flicker between black and white 37.5 times per second.

In order to free up CPU time and resources when performing common graphics routines, we will build a simple Graphics Processor (GP). For each frame of video output the CPU will write a list of graphics commands to data memory, and cue the GP to start drawing them. The graphics processor will read these graphics commands out of the processor’s data memory directly (using Direct Memory Access, or DMA), and draw the corresponding elements into the frame buffer. Your final GP will support at least four types of commands: LINE, FILL, STOP, and a new command of your choice. The specifics of the graphics commands are outlined in Graphics Command section later in the document. Both the MIPS 1

---

1 Check out the Nvidia web page for examples of more modern graphics processor languages and implementations.
and the GP will be able to write to the frame buffers, but for most of you most or all of your pixels will be written to the frame buffer by the GP.

**Figure 1** Shaded blocks are what you need to implement for the project: Graphics Processor (including Frame Filler and Line Engine), Pixel Feeder, and the contents of DRAM. The DRAM request controller handles all read and write requests to DRAM. The Chrontel DVI chip and driver handle all of the timing necessary to drive the display, as long as the pixel feeder keeps the input FIFO full.

**Timeline and Deadlines**

This is the last assignment for the project, with no explicit deadline except the project deadline. In order to give you a little more freedom and flexibility to balance the remainder of your semester, the due date for the completed project is November 30th, 2012 @ 3:00PM in the lab. It is your responsibility to manage your time. All other checkpoints are also due November 30th.

There is no late credit for any labs, or project checkpoints after this time-space coordinate. All repositories must be appropriately tagged and pushed to the remote repository by that date. Any commits that are pushed to the Github repository after the due date will be ignored.

**SDRAM**

Our SDRAM module contains 256MB organized as 32M words of 8B each. The words can be read or written on both the rising and falling edge of the clock 200MHz SDRAM clock, and the chips allow burst read and write lengths of four. As a result, the most time-efficient transfer block size is 256b=32B, and this is what the SDRAM controller implements. Running at 200MHz with double data rate, these 32B quantities enter and leave the SODIMM in 10ns. The DRAM Request Controller has a FIFO interface to the DRAM that is 128b=16B wide, and all reads and writes operate on 128b quantities. To match this with the SDRAM block size, two 128b transfers happen on both reads and writes through the DRAM Request Controller.
For writes to SDRAM, a 16b write_mask is passed along with the 16B of data to indicate which bytes to write. A “1” in the write_mask means that the corresponding data byte is masked (not written). If you want to write all bytes, the write_mask should be all zeros.

**DRAM Request Controller**

The Request Controller is basically an oversized multiplexor that manages data flow between all the modules that need to access DRAM and the DRAM FIFO using a ready/valid interface. Because the DDR2 operates on a different clock rate than our CPU, all the requests need to go through a single clock crossing FIFO. This is what the Request Controller multiplexes access to.

The Request Controller handles each of the requests that it receives through some priority logic as some requests are more important. The direction of the data flow is illustrated in the block diagram shown earlier. You will notice that some modules read xor write to DRAM, and some modules both read and write. You do not have to worry about starvation for any of the modules requesting data from the FIFO.

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.
5. Assert wdf_wr_en and af_wr_en when !af_full && !wdf_full.
6. Supply the next 128 bits of data and assert wdf_wr_en when !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 rst 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.
Frame Buffers

The frame buffer is simply a region in memory where pixels are stored. You will be using (at least) two frame buffers and switching between them, often referred to as double-buffering. The addresses of the frame buffers are up to you, but 0x1040_0000 and 0x1080_0000 might be convenient.

The addressing scheme for the frame buffer is also your choice, but a convenient approach is to use 10 bits for the pixel y coordinate and 10 bits for the pixel x coordinate. Using them in y,x order in the address means that pixels on the same row are stored sequentially. Thus in order to write to pixel (x, y) in the first frame buffer 0, assuming a base address of 0x1040_0000, we need to write to address:

\[
\text{address} = \{10'b0001_0000_01, y, x, 2'b0\}
\]

Likewise, if we wanted to write to the second frame buffer we would simple use:

\[
\text{address} = \{10'b0001_0000_10, y, x, 2'b0\}
\]

The output resolution of the screen that we will be using is 800 x 600 pixels and the base address of the frame buffer corresponds to the top left pixel (or the origin) of the display. The scheme outlined above does leave some portions of the memory in the frame buffer unused but greatly simplifies the address scheme. You will need to keep this in mind when you write your FrameFiller.v and PixelFeeder.v modules.

If you need a quick review of frame buffers, you can consult the following slides:


PixelFeeder

The first step to displaying pixels on the screen is to complete the PixelFeeder.v module. This module is responsible for continuously reading from the frame buffer address space of the DRAM and feeding the pixel values to the DVI driver.

The DVI driver causes the Chrontel DVI chip to generate SVGA 800x600 at 75 Hz. You won’t need to know many of the details, since the DVI driver module is given to you, but some of the timing may be useful to you in other parts of the project. Details are available here: http://tinyvga.com/vga-timing/800x600@75Hz

We have provided the FIFO and output logic of the module in PixelFeeder.v, all you need to do is fill in the logic that will keep the FIFO from running empty. The primary challenge will be dealing with the DRAM latency. When you submit a request you will need to make sure that there is room on the FIFO when the request returns from the DRAM. To do this, we recommend counting the number of pixels requested and in the FIFO.
Keep in mind that your PixelFeeder.v module must also switch between buffers after drawing one complete buffer. This is done by requesting pixels from the appropriate DRAM frame buffer address depending on which frame buffer you are currently reading from.

Your task is to complete the PixelFeeder.v module using an FSM that keeps the FIFO as full as possible. If you are able to get the module to work, you should see the colors of the RGB rainbow displayed on the display that is plugged into the DVI output of your FPGA.

When the pixel feeder pulls the last pixel of a frame out of SDRAM it should raise an interrupt to let the processor know that it is done. At this point, the ISR that handles the interrupt can change the Pixel_frame register to point to the next frame buffer to be drawn, and change the GP_frame register to point to the next register to draw into.

**Graphics Processor**

The CommandProcessor.v module is responsible for reading the command list from a designated place in memory given by a memory mapped register, GP_CODE, and writing to a designated frame buffer, given by memory mapped register GP_FRAME. A write to GP_CODE will cause the GP to start reading and executing graphics commands at the address in GP_CODE, and writing to a frame with pixel[0,0] in location GP_FRAME. When the Graphics Processor sees a write to GP_CODE, it will immediately begin fetching and processing commands that live at that address in the DRAM. For each instruction, the Graphics Processor will check the type of instruction and drive the appropriate graphics engine until it reads a STOP command.

The Graphics Processor should generate an interrupt when it is done processing a command list. The CPU may choose to ignore this request (set the corresponding interrupt mask bit to 0). If the CPU performs a write to GP_CODE while the GP is still processing a previous command list, the GP should stop processing the old list and start processing the new list. The GP should have a memory mapped address register that the CPU can check to see the address of the current or next command (your choice) that the GP is processing. This could just be the GP_CODE register.

The Pixel Feeder and the Graphics Processor both share access to the same DRAM as the MIPS processor. As a result, they must be able to deal with read stalls just like the MIPS processor.

**Command List**

You will implement at least four different commands in the Graphics Processor. Commands consist of one or more 32 bit words. The most significant byte of the first 32 bit word is the command type. The remaining bits may be arguments, or don’t cares. The LINE command consists of three 32-bit words, which contain the color of the line as well as the two endpoints.

STOP  0x00, 0x000000

FILL  0x01, Red[23:16], Green[15:8], Blue[7:0]
The format of the points in the line command is chosen so that it will be easy for MIPS software to generate, and easy for humans to read in hex. When you write to the SDRAM, you will need to massage the X and Y values a bit. Since writes to SDRAM are four pixels (16B wide), the least significant four bits of the address will always be zero (4 Bytes/pixel). The two least significant bits of the X pixel address determine where in the 16B data the values go, and do not show up in the address supplied to SDRAM.

You will need to add at least one more command type for full credit on your project. Some possibilities include drawing pixels, text, triangles, circles, boxes, shaded lines, etc.

**STOP Command**

When the STOP command is reached, the Graphics Processor should raise an interrupt request and return to an idle state to wait for the next write to GP_CODE.

**Frame Fill Command**

The frame fill command will simply fill the entire frame with the RGB color specified in the color payload. The Graphics Processor will do this by calling the FrameFiller.v module as soon as it is ready to accept a new frame fill request. Writing to the frame filler will automatically trigger a frame fill operation upon a handshake. The module will also take in the target frame buffer to operate on.

**Line Command**

The line command consists of three 32 bit words. The first contains the desired color of the lines. The second two contain the two endpoints. The Graphics Processor should deliver these arguments, and the target frame address, to the Line Engine module.

**Example**

if we want to clear the screen to black, and then write a blue line from (0x10,0x20) to (0x1A,0x2B), and a red line from (0x123,0x124) to (0xAA, 0xBB), the graphics code in memory would look like:

```
0x4000:  0x0100_0000     # black fill
0x4004:  0x0200_00FF     # blue line
0x4008:  0x0010_0020     # first endpoint
0x400C:  0x001A_002B     # second endpoint
0x4010:  0x02FF_0000     # red line
0x4014:  0x0123_0124     # first endpoint
0x4018:  0x00AA_00BB     # second endpoint
0x401C:  0x0000_0000     # STOP
```

To get the graphics processor to execute these instructions, the MIPS programmer first loads the GP_FRAME register with the desired frame location, and then loads 0x4000 into the memory mapped
register GP_CODE. The graphics processor reads the first word at 0x4000, sees that it's a fill command, and calls the fill engine with the color black. Once that operation is done, the GP reads the next instruction (at 0x4004) and sees that it's a line with color blue, grabs the next two words to determine the endpoints, and then uses the Line Engine (described below) to draw the line in SRAM. Then it does the same with the red line. When it's done with the red line, it reads the STOP instruction, requests an interrupt, and returns to an idle state, waiting for a signal from software to begin drawing the next frame.

If done properly, the frame fill should take 480,000 cycles, and the two lines a few hundred more.

Frame Filler

The frame filler is responsible for accelerating the frame filling rate and lives in the FrameFiller.v module. When you execute a software fill from the BIOS, you will notice that the filling is slow enough that you can see it progress across the screen. To the CPU, waiting for the frame to fill is an eternity and burns processor cycles.

Your job is to implement the logic and FSM control that will fill the target frame buffer with the same color. The goal is to beat a well-written software frame filler.

Line Engine

Bresenham’s line algorithm is nice because it uses only integers, and doesn't use multiplication or division. You will find many implementations on the web, some of which may be more relevant than others. Here’s one in C from Wikipedia:

```c
#define SWAP(x, y) (x ^= y ^= x ^= y)
#define ABS(x) (((x)<0) ? -(x) : (x))

void line(int x0, int y0, int x1, int y1) {
    char steep = (ABS(y1 - y0) > ABS(x1 - x0)) ? 1 : 0;
    if (steep) {
        SWAP(x0, y0);
        SWAP(x1, y1);
    }
    if (x0 > x1) {
        SWAP(x0, x1);
        SWAP(y0, y1);
    }
    int deltax = x1 - x0;
    int deltay = ABS(y1 - y0);
    int error = deltax / 2;
    int ystep;
    int y = y0
    int x;
    ystep = (y0 < y1) ? 1 : -1;
    for (x = x0; x <= x1; x++) {
        if (steep)
            plot(y,x);
        else
```
plot(x,y);
error = error - deltax;
if (error < 0) {
    y += ystep;
    error += deltax;
}
}

In this case, plot(a,b) would mean a write to pixel (a,b) with the current color in the current frame buffer. Your job is to translate this C code into hardware, and generate one pixel per clock cycle.

Making things move

To draw things that move on the screen, you will need to erase the frame buffer in DRAM (i.e. write black pixels) at the end of each frame before you start to draw the new one. To do this, the processor needs an interrupt at the end of each frame to tell it that it's time to erase that one and get started on the new one. This interrupt is called the video blanking interrupt, or VBI. The VBI should be requested when the video engine is done reading the frame. It's easy to figure out when the frame is done: just watch the address lines coming out of the video engine.

The video blanking interrupt service routine (VBISR) doesn't have to do much: tell the pixel feeder to pull from the new frame, and tell higher level software that it's time to define the commands for the next frame after that. Probably the best way to do that is to increment a frame counter that both watch.

It's up to you to decide how to erase the frame. You can put a frame fill command at the beginning of each GP instruction list. Or you could have your software use the graphics processor to erase each graphics element that it drew in the previous frame, before starting to draw new ones.

Telling the graphics processor to start drawing the new frame is easy. The graphics processor should start drawing when you write to the GP_CODE register. The act of writing that register triggers the graphics processor to start executing the code at that location. So in our example above, if we just wrote 0x4000 to GP_CODE once, we'd get exactly one screen with a red and blue line on it, and then it would disappear in about 13ms (our frame rate is 75Hz).

The higher level software gets the signal that a new frame is started, and puts graphics instructions into a different memory location in preparation for the next frame. For example, if we take the graphics processor instructions above, and make a copy of them at a new location (0x5000), and increment the x values of the blue line by 1, and the y values of the red line by 1:

```
0x5000:  0x0100_0000     # black fill
0x5004:  0x0200_00FF     # blue line
0x5008:  0x0011_0020     # first endpoint - x has moved right
0x500C:  0x001B_002B     # second endpoint
0x5010:  0x02FF_0000     # red line
0x5014:  0x0123_0125     # first endpoint - y has moved down
0x5018:  0x00AA_00BC     # second endpoint
0x501C:  0x0000_0000     # STOP
```

then the next time the VBISR runs it can write 0x5000 into the pixel feeder, and the blue line will move to the right one pixel, and the red line will move down one pixel.
Here's an example of what the high level code might look like:

```c
while(1) {
    if (frame%2) {
        writeOddFrameGPInst(0x5000);
        GP_FRAME = 0x10800000;
        GP_CODE = 0x5000;
    } else {
        writeEvenFrameGPInst(0x4000);
        GP_FRAME = 0x10400000;
        GP_CODE = 0x4000;
    }
    oldframe=frame;
    while (frame == oldframe) ; /* spin waiting for VBISR to increment frame */
}
```

When `frame` is even, the pixel feeder will draw from the "even" frame buffer (0x1040_0000 in this case), and software will be writing instructions that the Graphics Processor puts into the "odd" frame buffer (0x1080_0000 in this case). It's more efficient to put in even one more "software pipeline" stage here, with the pixel feeder pulling frame N from one frame buffer, the graphics processor drawing frame N+1 in the other frame buffer, and software writing commands for frame N+2 into a command buffer, but that's up to you.

There are also better ways to wait than idle spinning, but if our game inputs (keyboard, etc.) use interrupt service routines to change the game state, and the write*FrameGPInst() functions read that state, then this works.

### Deliverables

None, other than what you need for the final project.

### Suggestions

Divide and conquer. Start off with a hardcoded frame buffer address in your pixel feeder. Using just software, write some simple patterns and colors into that single frame buffer in DRAM, and make sure that they show up on the display. When that works, add in the ability to write to the Pix_Frame register to tell the PF where to get its frame. Write something different into a second frame buffer, and make sure that you can switch between them. Maybe use your 1 second timer, or use your UART_RX_ISR.

When implementing the line drawing engine, first make sure that you can draw the two endpoints, then a line from (x0,y0) to (x1, y0) rather than (x1,y1), so that you just need the X counter and not the Y math. When all that works (save it!) then do the rest.

The BIOS has some software drawing routines that may help you debug. They are

```
swline <6 digit hex color> <x0> <y0> <x1> <y1> <buffer>
swfill <6 digit hex color> <buffer>
```

to draw a line or fill a buffer. These will only work if you use the frame buffer addresses 0x1040_0000 and 0x1080_0000.
Questions

1) Referring to the SVGA timing website for our rate, answer the following questions.
   a. 800x600*75=36M pixels per second. Why is the pixel clock on the Chrontel 49.5MHz?
   b. How much time does it take to draw a complete line (first pixel to last pixel)?
   c. How much time is it from drawing the very first pixel until the very last pixel is drawn on the screen? If you used a single frame buffer, how much time would you have to fill in the frame buffer between the end of one frame and the start of the next?

2) If your first frame buffer address is 0x1040_0000,
   a. What is the address of the first pixel of the first line? Last pixel of the first line? First pixel of the second line? First pixel of the last line? Last pixel of the last line?
   b. If your second frame buffer address is 0x1080_0000, how much room is there between the last pixel of the first frame, and the first pixel of the second frame? How many line commands could you fit in that much space?

3) If the Pixel Feeder is the only module accessing SDRAM and is pulling pixels out at top speed,
   a. What is the total worst-case amount of time between subsequent (back to back) requests for a block of 8 pixels?
   b. What is the total amount of time required to pull all of the pixels needed for a single frame? Can any other read requests be serviced during this time? Write requests?
   c. What fraction of the total SDRAM read bandwidth will the Pixel Feeder use? Write bandwidth? Why are these different?

4) If the MIPS processor could approach one pixel per clock cycle for a software frame fill, and assuming that the Cache write-through could keep up (no stalls) how long would it take to fill an entire frame in software?

5) Knowing that in fact each pixel that the MIPS writes to the Request Controller causes a full SDRAM row read/write/precharge, how long will the software frame fill take?

6) If the Frame Filler is the only module accessing SDRAM, what is the time required to fill an entire frame buffer?

7) If you were to redesign all of the logic between the Frame Filler and the SDRAM to optimize the speed of frame filling,
   a. Describe (draw) the sequence of opening a row, writing to columns, precharging, and opening the next row.
b. How many clock cycles would it take to write a single row using a Burst Length of 4? Of 8?

c. How fast could you fill a single frame?

8) If you were to redesign the logic between the Frame Filler, the Pixel Feeder, and the SDRAM, and if you were to put your two frame buffers in separate SDRAM banks

   a. Describe (draw) the sequence of opening rows, reading and writing to columns, precharging, etc. that would minimize the total number of cycles required to fill one frame while feeding another to the DVI driver.

   b. How many clock cycles would it take to accomplish this? Would it be optimal to use a Burst Length of 4, or 8?

   c. Could the DVI driver FIFO, currently 8192 pixels deep, keep up?

9) If the pixel FIFO is full when the Pixel Feeder pulls the last pixel of a frame out of SDRAM, how many microseconds will it be between the PF_IRQ signal and the time that the last pixel of that frame is drawn on the screen?