EECS150: Fall 2010 Project Checkpoint 3, Graphics Engine

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

Revision D

1 Time Table

ASSIGNED	Nov 12
DUE	Week of 11/29

2 Overview

Now that the basic infrastructure for the video subsystem is complete, we can display an entire frame of video on the screen. Still, however interesting the frame is, it is just a single frame (a static image!). In order to make the video subsystem useful, we must have a means of changing the image that is to be displayed, on a frame-by-frame basis.

As we saw in Checkpoint 2, it is easy to change a single image with software. Software changes the contents of the frame buffer through the MIPS150 processor. For example, to clear the entire screen, we could perform 480,000 **sw** operations (one for each pixel on the screen) and write the same color into all locations. Similarly, lines, shaded polygons, and anything else can be drawn with software algorithms. This works fine, but it takes a lot of CPU time on tasks that are often done much faster in hardware.

To make things move on the screen, we need them to "move" in the SRAM. This means that at the begining of each frame we need to get rid of the things that aren't in the same place. This means either "undrawing" each graphics element from the previous frame, or simply clearing the entire frame in SRAM, and starting from scratch with drawing the new one in SRAM.

In order to free up CPU time and resources when performing common graphics routines, we will build a simple **Graphics Processor**. The graphics processor will read simple graphics commands out of the processor's data memory directly (using Direct Memory Access, or DMA). The graphics processor will have a memory mapped control register, GP_CODE, that tells it the starting address of it's instructions. Initially there will only be two commands that the graphics processor understands: STOP, and LINE. Each graphics instruction is one or more 32 bit words, inst[31:0]. The format of graphics instructions is an 8 bit command TYPE in the most significant byte, inst[31:24], and optional arguments in the other 3 bytes. The STOP command is TYPE=0. The LINE command is TYPE=1, with a 16 bit color as the least significant bits of the instruction, inst[15:0]. The LINE command is followed by two additional 32-bit word arguments. The first argument contains the x,y pixel location of one endpoint of the line, the second argument contains the x,y pixel location of the other endpoint of the line. x,y pixel locations are 16 bits each (for forward compatibility with future high-res displays), even though we'll only use 10 bits of each. In other words, a 32 bit x,y location will be stored as 0xXXXX_YYYY, where for us 0<XXXX<799 and 0<YYYY<599.

At the beginning of each frame, the graphics processor will start reading commands at the location stored in GP_CODE, and keep executing them until it reaches a STOP command. Reading a single command, such as a line from one side of the screen to the other, might result in the graphics processor spending many hundreds of cycles to execute the command. Other commands, like a very short line, might take only a few cycles.

So to draw a collection of static lines on the screen, the MIPS programmer would pick an address in MIPS data memory, call it "GPcmds", write the GP commands for the desired lines starting at that address, finish the list with the STOP command, and then write the value GPcmds into the memory mapped register GP_CODE. For example, if we want to 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_001f     # blue line
0x4004:  0x0010_0020        # first endpoint
0x4008:  0x001A_002B        # second endpoint
0x400C:  0x0100_f800     # red line
0x4010:  0x0123_0124        # first endpoint
0x4014:  0x00AA_00BB        # second endpoint
0x4018:  0x0000_0000     # STOP
```

To get the graphics processor to execute these instructions, the MIPS programmer loads 0x4000 into the memory mapped register GP_CODE. Then the graphics processor reads the first word at 0x4000, sees that it's a line with color blue, grabs the next two words to determine the endpoint, 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, and stops doing anything, waiting for the beginning of the next frame.

3 VBI, VBISR

To draw things that move on the screen, you will need to erase the SRAM (i.e. write black pixels) at the end of each frame before you start to draw the new one. If we had more video memory, we'd probably double buffer the video (draw in one frame while displaying the other, then switch), but we don't. So we'll use an old trick from the early video games where the processor gets an interrupt at the end of each frame to tell it that it's time to get started on the new one. This interrupt is called the vertical 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 vertical blanking interrupt service routine (VBISR) doesn't have to do much: erase the old frame from SRAM if it isn't already, tell the graphics processor to start drawing the new frame, and tell higher level software that it's time to define the commands for the next frame.

It's up to you to decide how to erase the frame. One way is to have software do it at the end of each frame. That's fine, but you can do much better. You can erase the frame as the video engine is reading it out - just write back 0x0 into each 32 bit pixel pair that the video engine has read. Or you could use the graphics processor to erase each graphics element that it drew in the previous frame.

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 one:

```
0x5000:  0x0100_001f     # blue line
0x5004:  0x0011_0020        # first endpoint
0x5008:  0x001B_002B        # second endpoint
0x500C:  0x0100_f800     # red line
0x5010:  0x0123_0125        # first endpoint
0x5014:  0x00AA_00BC        # second endpoint
0x5018:  0x0000_0000     # STOP
```

then the next time the VBISR runs it can write 0x5000 into GP_CODE, and the blue line will move to the right one pixel, and the red line will move up one pixel. If the higher level software then writes new values into the graphics processor code in location 0x4000, there will be instructions ready for the next VBISR.

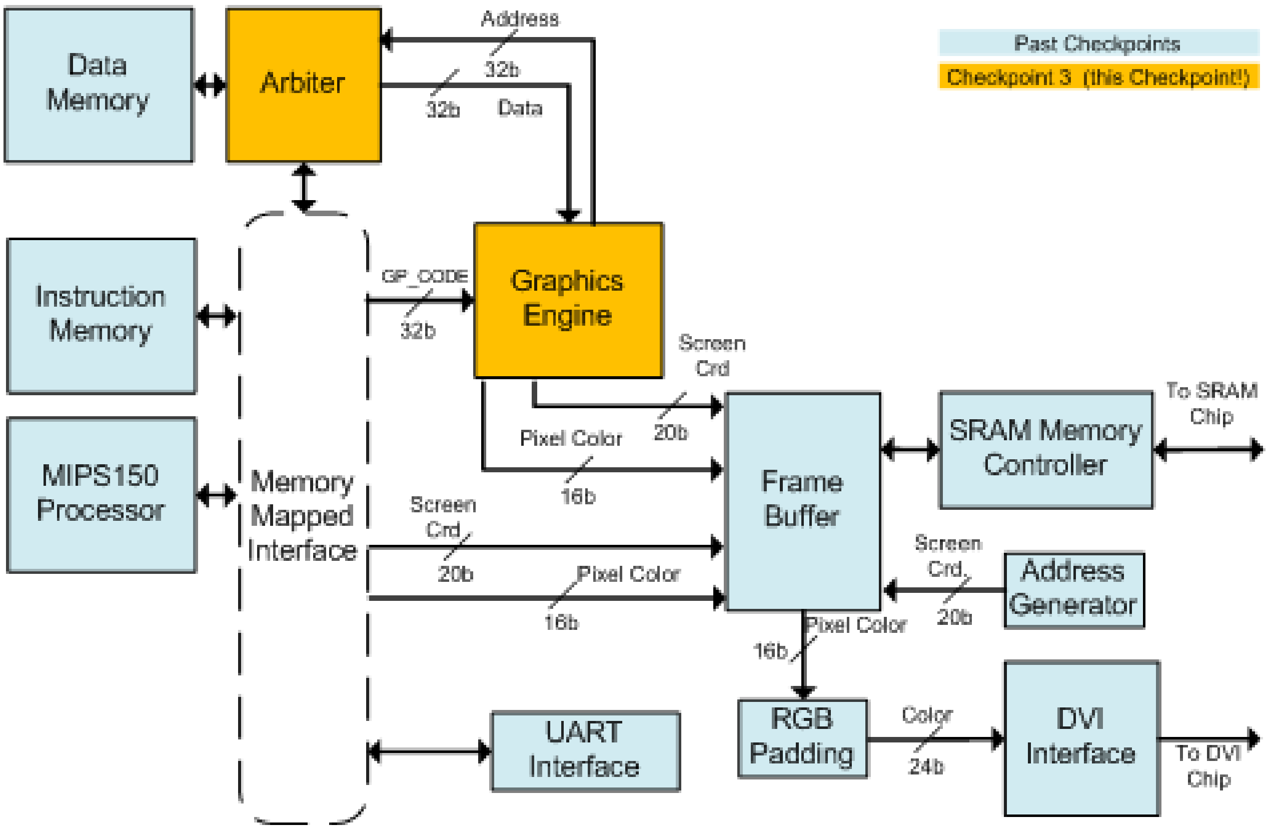
Here's an example of what the high level code might look like:

```
while(1) {
  if (frame%2) {
    writeOddFrameGPinst();
    GP_CODE = 0x5000;
  } else {
    writeEvenFrameGPinst();
    GP_CODE = 0x4000;
  }
  frame++;
  newframe=0;
  while (!newframe) ; /* spin waiting for VBISR to set newframe; */
}
```

Note that there are better ways to wait than idle spinning, but if our game inputs (keyboard, N64 controller) use interrupt service routines to change the game state, and the write*FrameGPInst() functions read that state, then this works.

4 Line drawing engine

Figure 1 Your project circa checkpoint 3.



The first feature of this graphics engine will be the Line Engine that works alongside the processor and frame buffer. The job of the line engine is to quickly draw a line from one set of x, y coordinates to another. This block will not only speed up the line drawing process but also allow for the CPU to be doing other work while lines are being drawn.

We will be using the **Bresenham Line Drawing Algorithm** to to implement the the line engine. The algorithm is explained below using C. Your job will be to create a hardware implementation for the given software routine.

5 Checkpoint Components

Your project, factoring in work done for checkpoint 2, now looks like what is shown in Figure 1.

Your goal in this checkpoint will be to implement the graphics processor in hardware, the VBI hardware, and the VBI interrupt handler. The graphics processor consists of several pieces: the DMA engine, the instruction parser, and the line engine.

As mentioned in Section 2, we have provided the software routine, that represents what your hardware implementation will do, in Program 1.

Program 1 Bresenham Line Drawing Algorithm in 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 - deltay;
        if (error < 0) {
            y += ystep;
            error += deltax;
        }
    }
}
```

The implementation shown in Program 1 works for drawing lines of any slope and for lines in any quadrant of the 2D plane. Looking at the software implementation, think about how the MIPS150 processor would perform the line drawing algorithm if it had to in software. How long would drawing an entire line take?

Debugging tip: Before you implement the full algorithm above, you might want to start by verifying that you can draw the two endpoints all by themselves.

6 Architectural Concerns

Your goal after implementing the line drawing algorithm shown in Program 1 is to write one new pixel to the frame buffer every cycle. Contrast this performance with that which you would be able to obtain from the MIPS150 processor running the algorithm in pure software. Know that in order to meet timing, you may have to pipeline your implementation.

One of the last things to consider is the fact that the Graphics Processor is not the only thing trying to access the SRAM. There will be times when the Graphics Processor is attempting to write during a cycle when the SRAM arbiter is not able to accept a write from the Graphics Processor. To support flexibility in memory arbitration, your Graphics Processor must also support the ability to stall. It should accept a signal, `stall`, that, whenever high, halts the operation of your Graphics Processor's pipeline. Line coordinates that are not able to be written must not be lost. The Line Engine should commit writes whenever it can.

Note also that the Graphics Processor is not the only thing that is accessing the MIPS data memory. With DMA, the Graphics Processor should only steal free data memory cycles, rather than stalling the MIPS.

You should design your Graphics Processor so that it's easy to add other instructions later. For example, drawing circles, filled polygons, etc. These will be extra credit on the final version.

Rev.	**Name**	**Date**	**Description**
D	Pister	10/12/2010	Modified to be consistent with the Fall 2010 project specification.
C	Kyle Wecker	4/17/2010	Modified to be consistent with the Spring 2010 project specification.
B	Chris Fletcher	4/21/2009	Clarified how the line engine starts an operation (added information about the difference between writing to address **0x8040_0040** and address **0x8040_0044**.
A	Chris Fletcher Ilia Lebedev	3/26/2009	Wrote new Document