The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 541 Digital Logic and Computer Design
Fall 2014

Lab Project (PART A): A Full Computer!
Issued Wed 11/5/14; Due Fri 11/14/14 (11:59pm)

You will learn the following in this lab:

- Designing a module with multiple memories
- Designing and using a bitmap font
- Designing a memory-mapped display
- Understanding initialization of a memory unit using an external file

---

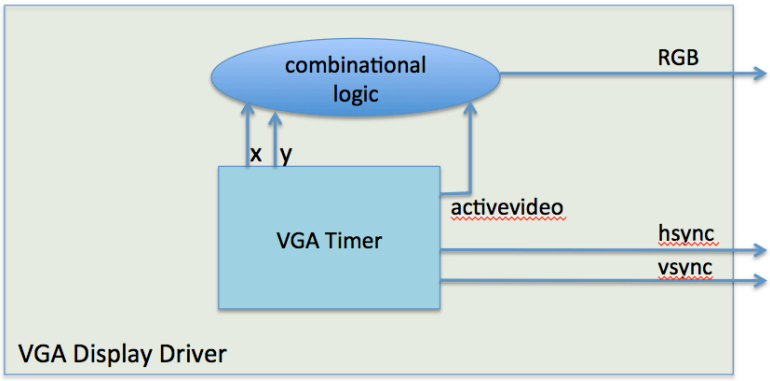
Part 1: Finish up your design of the MIPS CPU

Add any remaining instructions that you think you will need for your final demo. At a minimum, you will need to have the following implemented:

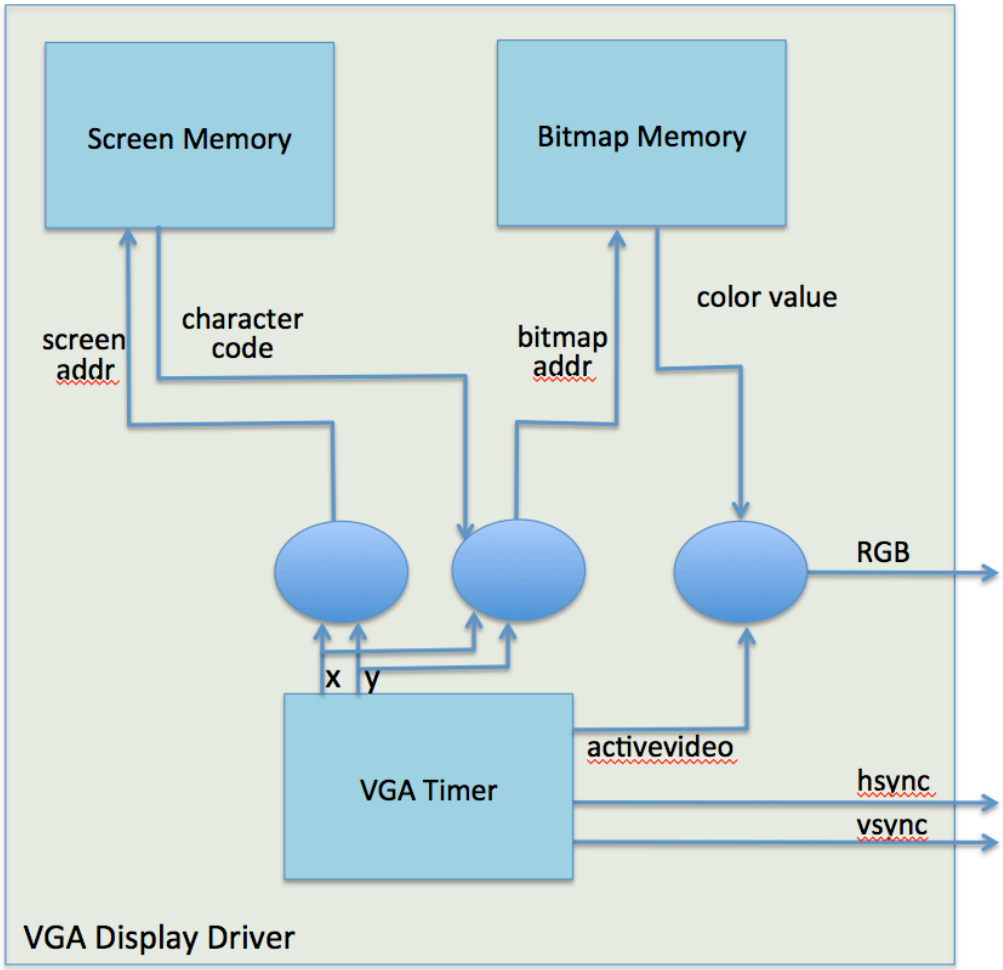
- Thoroughly test all of the instructions specified in Lab 10
  - `lw` and `sw`
  - `addi, slti, ori`
    - NOTE: While `addi` and `slti` should sign-extend the immediate values, `ori` should zero-extend the immediate because it is a logical operation!
  - R-type (all that are supported by our ALU, including shifts and the two comparisons)
  - `beq, bne` and `j`
- Add the following new instructions:
  - `jal` and `jr` (to implement procedure calls and returns)
    - These two instructions need slight modifications to the datapath, e.g.: adding another input to the multiplexer that selects the write address for the register file ("31" for `jal`); adding another input to the multiplexer that selects the write data for the register file ("PCPlus4"); adding another input to the multiplexer that selects how the PC is updated (for `jr`); etc.
    - See Slides #18 and #20 from Comp411 (Lecture: "Let's Build a Computer"), provided on the website. There may be some differences w.r.t. what you are designing, so use them as guidance only.

Part 2: Design a full display unit ("Terminal Display")

You will build upon your VGA display driver from Lab 7 to make it a full-fledged character display. The block diagram below shows your design from Lab 7:



This earlier design simply generated a fixed pattern to show on the display (e.g., lines, checkerboard pattern, etc.). In this assignment, you will extend it to display a 2-D grid of characters, i.e., a character display, as shown in the block diagram below.



The characters to display are assigned codes (your choice), and these codes are stored in an array in a special memory called *screen memory.* The array is stored in row-wise (row 0 first, then row 1, etc.), and left-to-right within each row. If your screen size is 640x480 pixels, and if you decide on each character being 16x16 pixels, then each row will have 40 characters, and there will be 30 rows. So, your screen memory will need to have at least 1200 locations.

There is also another memory, a read-only one, called *bitmap memory,* which stores the pixel pattern for each of the characters you implement. So, for example, if your characters are blocks of 16x16 pixels, and if each pixel has an 8-bit RGB color, then the bitmap memory will have 256 bytes stored for each character you choose to implement. If your final application needs 16 different characters, then your bitmap memory will have 256 x 16 = 4096 bytes of data.

Note: There is no CPU in this picture… yet.

Study the block diagram carefully. Make a top-level module called **`displayunit`** (in a file named **`displayunit.v`**) which contains three submodules:

- **A VGA timer:** You designed this module in Lab 7. You will need to modify it so that the color value is now calculated by looking up the two memories in sequence.
- **A screen memory:** This memory contains a linear sequence of values, each representing the code for a character. These could be ASCII codes if you would like, or codes you assign to some special characters (e.g., different colored blocks, or different types of smilie faces, etc.). It is sufficient to keep each code 8-bits wide, although you can use fewer bits if you need fewer. For instance, if you only want to display 32 unique symbols, you only need a code with 5 bits (your codes would run from 5'b00000 to 5'b11111). The width of the screen address will depend on your screen resolution and character size. If, as in the text above, your screen has a total of 1200 characters, you will need a screen address of 11 bits. *Note:* Each location in this memory should represent a single character's code.
- **A bitmap memory:** This memory is indexed by the character code, and stores the bitmap or "font" information for that character. In particular, each character is a 2-dimensional matrix of RGB values, stored in a linear sequence. So, for example, if each character is a 16x16 square box of pixels, you will store the 8-bit RGB value for the (0,0) pixel for that character first, then (0,1), and so on until the end of the top row, then the second row, etc. Thus, there will be 256 color values stored for each character. *Note:* Each location in this memory should represent a single pixel's color value.

Keep the following points in mind as you do this assignment:

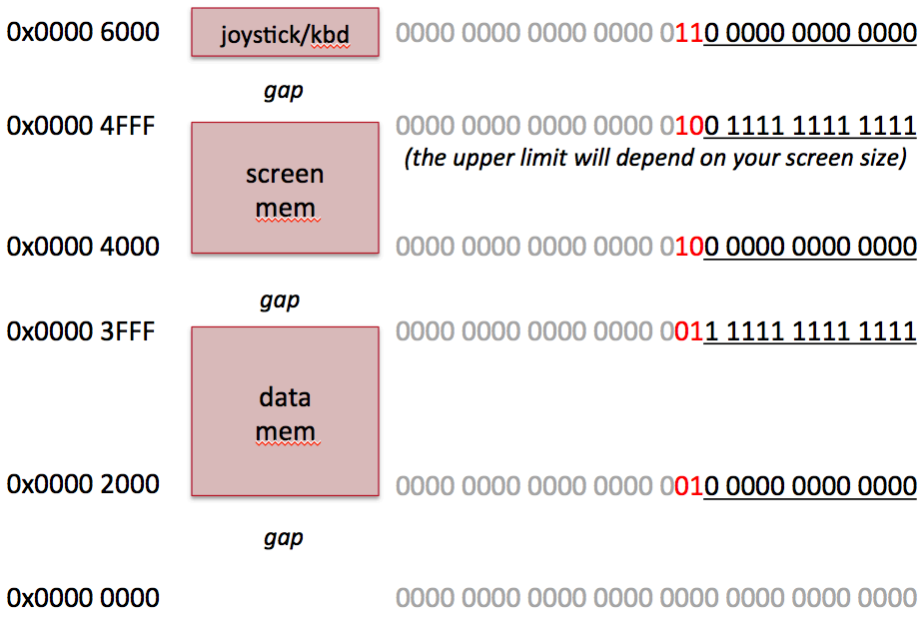
- Start with only a small number of characters (say, 2 or 4). If your design works fine, increase the number of characters to a reasonable number (at least 8, but as many as you think you might need to do an interesting demo!). You may have to think a bit into your final demo here, but don't worry, once you have the basic design working, it won't be too hard to come back and add more characters and bitmaps to it!
- Initialize the screen memory from a file using the **$readmemh** instruction. You should have the entire screen initialized in this file; otherwise there may be "junk" character codes in the part of the screen left uninitialized.
- Initialize the bitmap memory from a file using the **$readmemh** (or perhaps **$readmemb** may be more convenient here). If, for example, characters are 16x16 pixels, then each will require 256 color values to be stored in this memory. Start with only a couple of characters, then increase the number.

- The main challenge in this lab assignment is to instantiate the two memory units, and to wire everything up together. This is a good exercise in hierarchical design. That is the reason I will not be providing a Verilog code skeleton.
- The key challenge to designing this system is to figure out the following mappings:
    - The mapping from the *(x,y)* pixel coordinates generated by the VGA Timing Generator, to the *(J, K)* character coordinates that that pixel maps to in Screen Memory.
    - The mapping from *(J, K)* character coordinates to the address in Screen Memory.
    - The mapping from the character code that the Screen Memory gives you, to the start location of the bitmap stored for that character in the Bitmap Memory.
    - The mapping from the *(x,y)* pixel coordinates generated by the VGA Timing Generator, to the offset within the bitmap for that character in the Bitmap Memory.

Start small. If your design is too big, it might not fit onto the FPGA chip on our boards.

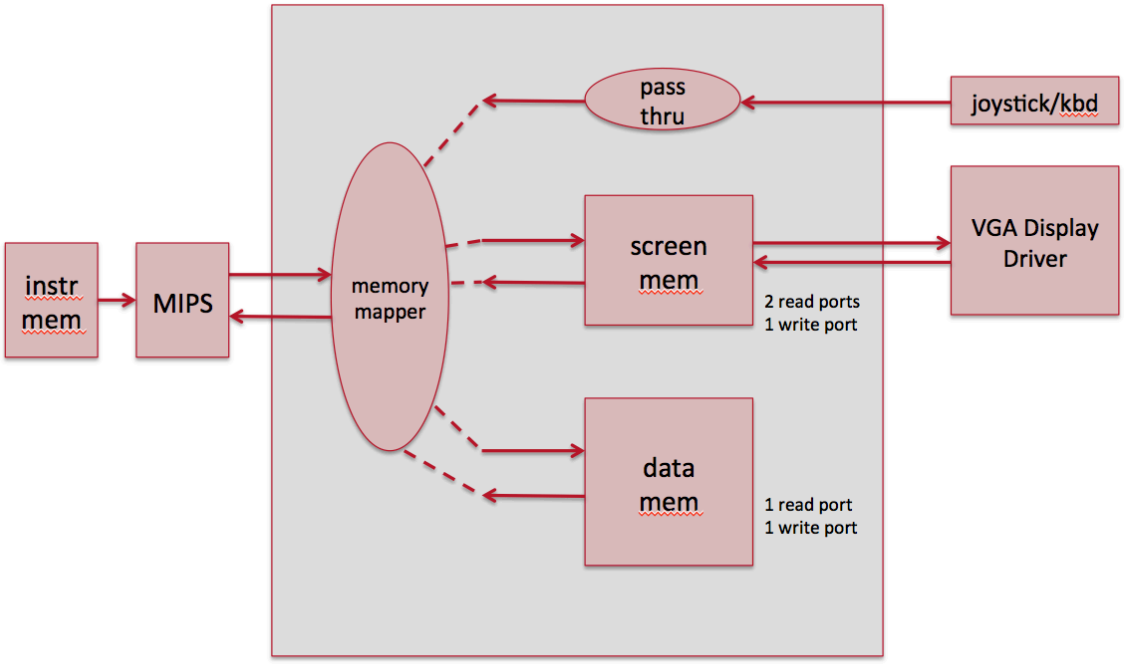
Part 3: Integrate the CPU and the display unit using memory mapping

As discussed in Lecture 16 (slides 3 and 5 [reproduced below]), you will integrate the CPU and the display using memory-mapped I/O. One possible memory mapping scheme was discussed in class:



Assigning the data memory to start at address 0x2000 allows you to use the MARS assembler with the "Compact, Text at 0" configuration, which places code at 0x000 and data at 0x2000.

To implement this memory map, use the block diagram below. Put the "Memory and I/O unit" in a module called **memIO,** and name the file **`memIO.v`**. For now, skip the joystick/keyboard etc. We will discuss those next week.



Memory and I/O Unit

NOTE: The screen memory was inside the display driver in Part 2 has now been pulled out and placed inside the "Memory and I/O unit". Therefore, the display driver now outputs the address for the screen memory, which goes into the memory-I/O unit through a port (shown on the right side in the figure above). This port is distinct from the port used by the MIPS processor, and from the port that will be used by the keyboard/joystick.

Implement this part on the boards! Write a short program to have your MIPS write characters to *Screen Memory* and see if they show up on the monitor! If all goes well, you will have a near-final full-function computer.

Good luck!

Start working on your final project demo. More on this next week.

What to submit:

- **[Part 1] A list of all the instructions you have correctly implemented. Also include a list of any instructions that you are still trying to implement, or plan to.**
- **[Part 2] The Verilog source of your top-level display unit (displayunit.v). This file should contain the Verilog description of the three combinational logic blocks for generating RGB values for each pixel.**
- **[Part 2] In a couple of sentences, describe the set of characters you have chosen to implement.**
- **[Part 3] The Verilog source for the memIO module (memIO.v). In a couple of sentences, state if everything works as you expect, or if there are some problems you still need to resolve.**
- **Show a working demo of your design for Part 3 in the lab session on Friday, November 21.**

How to submit: **Please submit your work by email by 11:59pm, Fri, Nov 14, 2014, as follows:**

- **Send email to:** `comp541submit-cs@cs.unc.edu`
- **Use subject line:** **Lab Project PART A**
- **Include the Verilog files as attachments as specified above, and the rest in plain text.**
