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

Comp 541 Digital Logic and Computer Design
Prof. Montek Singh
Spring 2020

Lab #7: Memories: A Basic Datapath; and a Sprite-Based Display
Issued Mon 2/24/20; Due Wed 3/4/20 (11:59pm)

This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission instructions are at the end.

You will learn the following:

- Specifying memories in SystemVerilog
- Initializing memories
- Designing a multi-ported memory (3-port register file)
- Integrating ALU, registers, etc., to form a datapath

Part 0: Understand how memories are specified in SystemVerilog

Memory Specification

A typical single-ported RAM module is described in SystemVerilog as shown in the file **`ram.sv`**. The number of "ports" in a memory is the number of distinct read/write operations that can be performed concurrently. Thus, the number of ports typically is the number of distinct memory addresses that can be provided to the memory. A single-ported RAM takes a single address as input, and can perform a read or write (determined by a write enable signal) to that address. Below is the main part of a memory specification:

```
logic [Dbits-1:0] mem [Nloc-1:0];      // The actual storage where data resides
always_ff @(posedge clock)             // Memory write occurs on clock tick
    if(wr) mem[addr] <= din;           // … but only if write is enabled
assign dout = mem[addr];               // Memory read occurs asynchronously
```

Please refer to **`ram.sv`** for the complete description of the RAM module.

Memory Initialization

An uninitialized memory module contains junk (i.e., undefined values), although typically your board will initialize it to all zeros (or ones). You can, however, specify the actual values to be stored in memory upon initialization. This is done using the command **$readmemh** or **$readmemb**. The former command allows you to specify values in a file in hex format, while the latter uses binary format.

Add the following line in the register file module right after the line where the core of the storage is specified (i.e., right after "**logic [Dbits-1 : 0] mem [Nloc-1 : 0];**"):

```
initial $readmemh("mem_data.mem", mem, 0, Nloc-1);
```

(Remember to put the initialization line *after* the declaration of the logic type **`mem`**, and remember to replace **`mem`** with the actual name of your memory storage.)

The first argument to **`$readmemh`** is a string that is the name of the file to be read, line by line, during compilation and synthesis, and its contents are used to initialize memory values. The second argument is the name of the variable that is the memory storage. The last two arguments specify the range of memory locations. In this case, they start with 0, and go up to *Nloc*-1, but you are welcome to specify a subset of the range if you do not have data to initialize the entire memory.

Create the file `mem_data.mem` in the project folder (using an external editor), and add it to your project using *Add Source…* and selecting its type as *Memory File.* Add values, one per line, in hex. Do not prefix each value by 'h'. Thus, if your memory has 8-bit data, your initialization file may look like this:

```
05    // Comments are allowed
A0
C1
...
```

You can also use the binary version of the initialization (**$readmemb** instead of **$readmemh**). In that case, the file will have a sequence of binary values, one per line (no 'b' before though):

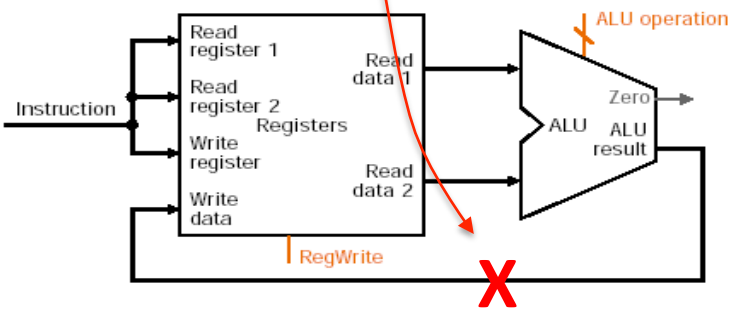
```
0000_0101  // Underscore can be used for clarity
1010_0000
1100_0001
…
```

Remember that if your datapath uses 32 bits, then the initialization values in `mem_data.mem` will have to be 32-bit values as well.

IMPORTANT: You must select the type of the file containing memory values as ***Memory File***. Otherwise, the tool may not be able to access it properly. If you did not choose the correct file type, you can right-click and change the file type.

Part 1: Register File

Today you begin to implement the MIPS processor that we will use for the final project. You will implement the portion of the CPU datapath shown in the following diagram, and test it. However, since we do not yet have a source for instructions, and to aid in testing, we will slightly modify this part of the datapath to provide more controllability and visibility. In particular, we will cut the feedback from the ALU to the write port of the register file, and instead allow *Write data* to be directly supplied by a test fixture. A modified picture is shown on the next page.



NOTE: We will NOT implement the datapath of this figure in this part. See figure on next page.

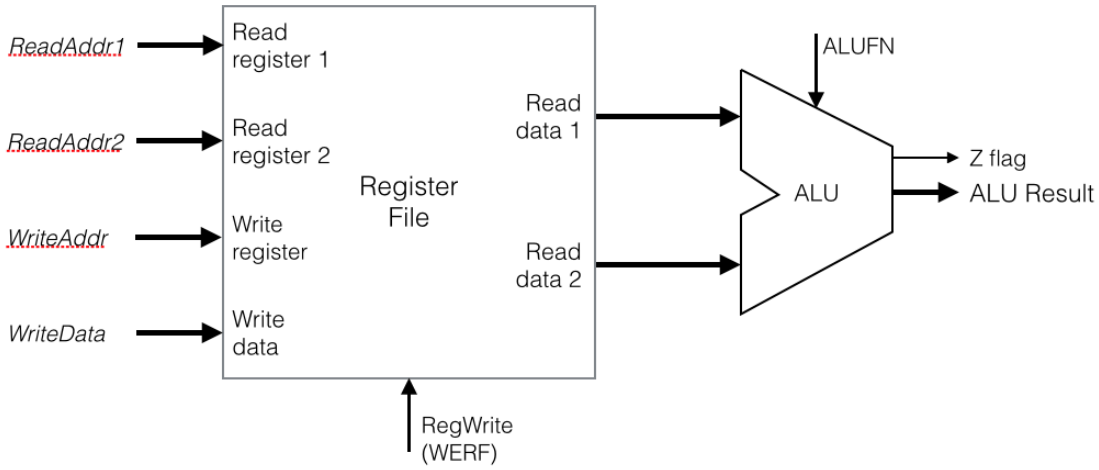
First, you will design a 3-port register file. We call it 3-port because three different addresses can be specified at any time: *Read Address 1, Read Address 2,* and *Write Addr.* These are required to access (up to) two source operands and one destination operand required for MIPS instructions.

Do the following:

- Start with the skeleton for a register file provided on the website (`register_file.sv`), and compare it to a typical RAM module (file `ram.sv`). The register file differs from the RAM memory in the following respects:
  - three address inputs instead of just one (e.g., *ReadAddr1, ReadAddr2* and *WriteAddr*).
  - two data outputs instead of just one (e.g., *ReadData1* and *ReadData2*).
  - the write enable and clock stay the same.
  - when writing, *WriteAddr* is used to determine the memory location to be written.
  - when *reading* register 0, the value read should always be 0 (it does not matter what value is written to it, or whether you write any values into it).
- Be sure to use parameters for number of memory locations (`Nloc`), number of data bits (`Dbits`), and the name of the file that contains initialization values (`initfile`).
- While in the final CPU design, the three addresses will come from the register fields in the instruction being executed, for now you will use a Verilog test fixture (in Part 2) to provide these addresses, the data to be written, and the *RegWrite* signal. The test fixture does a few different reads and writes so you can see via simulation that your register file is working.

Part 2: Putting the datapath together

Design a top-level module that contains the register file and your ALU (from Lab 2). Name the Verilog source `datapath.sv`. This module must exactly correspond to the block diagram below.

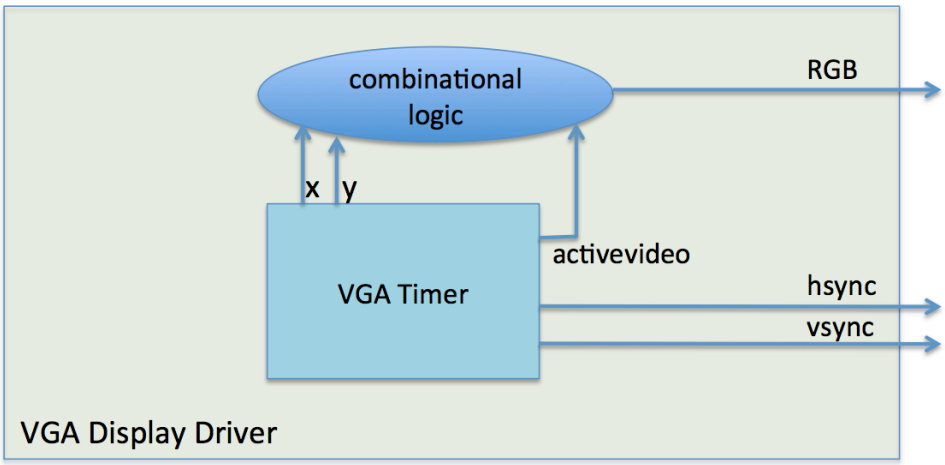


Note the following:

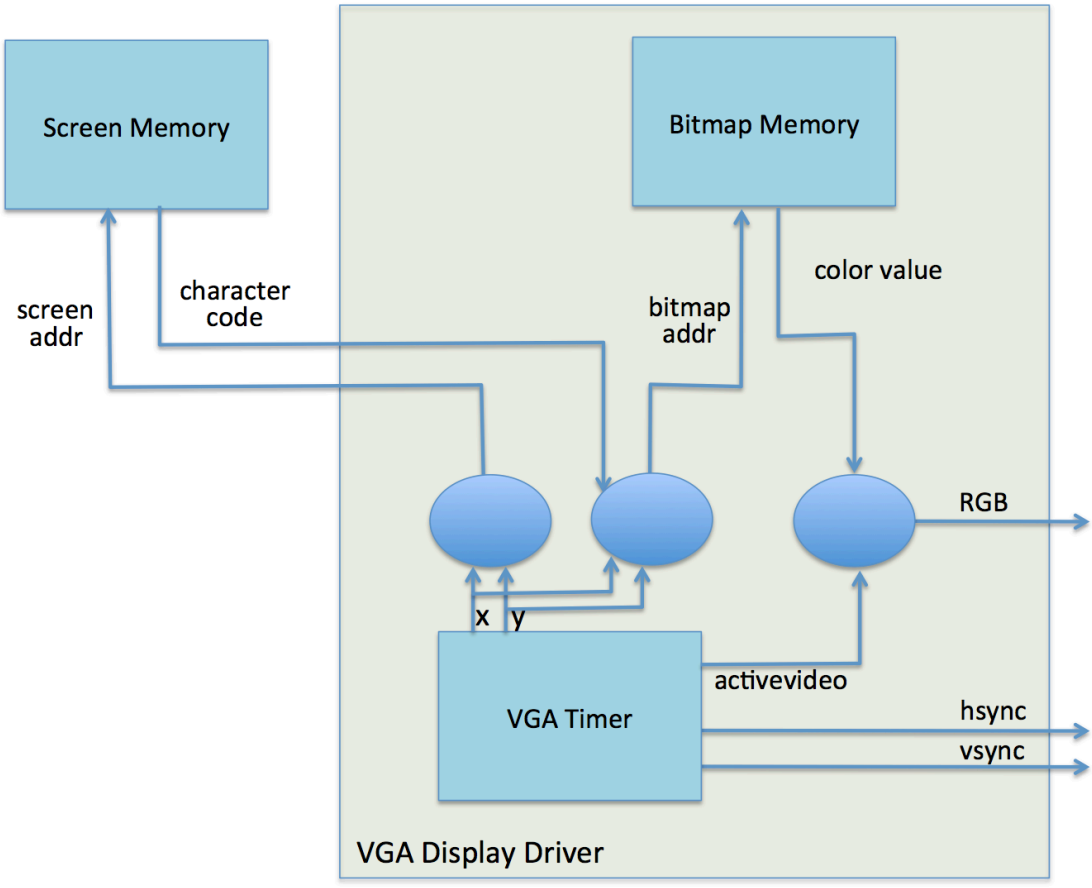
- To aid in testing your design, send "*ReadData1", "ReadData2"* and *"ALUResult"* to the output of the top-level module so they can be easily observed during simulation. The Zero flag (*Z*) must also be generated as an output from the top-level module (because branch instructions will need it).
- For now, do not feed the ALU result back to the register file. Instead, the data to be written into the register file should come in directly from the test fixture as an input to the top-level module.
- The inputs to the top-level module are: clock, *RegWrite*, the three addresses, the ALU operation to be performed (*ALUFN*), and the data to be written into the register file (*WriteData*).
- Use the Verilog test fixture provided on the website to simulate and test your design. The text fixture is self-checking, so any errors will be flagged automatically. Please use exactly the same names for the top-level inputs and outputs as used in the tester where the "unit under test" is instantiated.

Part 3: Design a full "sprite-based" display unit ("Terminal Display")

You will build upon your VGA display driver from Lab 4 to make it a full-fledged character-oriented terminal display. The block diagram below shows your design from Lab 4:



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 (or "sprites"), as shown in the block diagram below.



The characters to display are assigned codes, and these codes are stored in an array in a special memory called *screen memory*. The array is stored row-wise (row 0 first, then row 1, etc.), and left-to-right within each row. Let us set the size of each character at 16x16 pixels. If the screen size is 640x480 pixels, then each row will have 40 characters, and there will be 30 rows. So, your screen memory will need to have 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. Since each character is a block of 16x16 pixels, and each pixel has a 12-bit RGB color, the bitmap memory therefore should have 256 values (each 12-bit) stored for each character we implement. If your final application needs 16 different characters, then your bitmap memory will have 256 x 16 = 4096 values of 12 bits each. For now, let us implement 4 characters, to keep things simple.

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

Study the block diagram carefully. Make a top-level module called **`top`**, which contains **`vgadisplaydriver`** (in a file named **`vgadisplaydriver.sv`**) and **`screenmem`** (in file **`screenmem.sv`**). The VGA display driver in turn contains two submodules: VGA Timer and Bitmap Memory. Note the following:

- **VGA timer:** You designed this module in Lab 4. It supports a display resolution of 640x480. All of our design decisions will be based on this resolution.
- **Screen memory:** This memory contains a linear sequence of values, each representing the code for a character. These are codes you assign to some special characters (e.g., different colored blocks, or different emoji, etc.). The number of bits in each code will be determined by the number of characters used. For instance, if you want to display 32 unique characters, you will need a code with 5 bits (your codes would run from 5'b00000 to 5'b11111). It is best to parameterize this number. The number of bits needed for the screen address will depend on the total number of characters displayed on the screen. Since our screen has 1200 characters, we will need a screen address of 11 bits, with addresses in the range 0 to 1199. *Note:* Each location in this memory should represent a single character's code.
- **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. Since our characters are 16x16 square boxes of pixels, you will store the 12-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 (each 12-bit in length) 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 a total of only 4 characters. If your design works fine, you are welcome to add more characters (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 later.
- Initialize the screen memory from a file using the **$readmemh** instruction. You should have the entire screen initialized in this file; otherwise there may be "junk" character codes in the part of the screen left uninitialized. This file will be long (e.g., 1200 lines if your screen has 40 columns x 30 rows).
- Initialize the bitmap memory from a file using the **$readmemh** (or perhaps **$readmemb** may be more convenient here). For characters that are 16x16 pixels, each will require 256 color values to be stored in this memory. Start with a small number of characters (say, 4), then increase the number. This file can get long!

- 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 so things are easier to debug. Implement your design on the board. You should see the pattern described by the values in the screen memory and the bitmap memory (alternating red and green vertical bars).

What to submit:

- **[Part 2] Your Verilog source for the register file (register_file.sv) and datapath (datapath.sv).**
- **A screenshot of the simulation waveform window for Part 2 using the self-checking tester.**
- **[Part 3] Your Verilog source of the top-level display unit (vgadisplaydriver.sv). This file should contain the Verilog description of the three combinational logic blocks for generating RGB values for each pixel.**
- **A picture (phone photo is fine) of the monitor for Part 3 implementation.**

How to submit: **Please submit your work by email by 11:59pm, Wed Mar 4 as follows:**

- **Send email to:** `comp541-submit-s20@cs.unc.edu`
- **Use subject line: Lab 7**
- **Include the attachments as specified above**
