ENGR xD52: MP b011

Due November 22th 11PM EST

The purpose of this machine problem is to specify, create, and test a 32-bit CPU.

Work in groups of 3 to 4. This may require some team shuffling. One representative from each team must submit all deliverables electronically to comparch2013@gmail.com as a single compressed archive. Please format the email's Header as "[CA] [MP1] Name1 Name2 Name3"

The Work Plan

This subdeliverable is due within 36 hours as an email to comparch13@gmail.com with the subject line formatted as [CA][MP2 Work Plan] Name1 Name2 Name3". Any questions regarding this section should be directed to Molly Farison or Eric VanWyk.

Create a work plan for this lab. Break down the lab in to small portions, and for each portion predict how long it will take (in hours) and when it will be done by (date). You will be comparing your predictions to reality later.

The Basics

Create a 32-bit MIPS-like single cycle CPU that supports the following instructions:

LW, SW, J, JR, JAL, BNE, XORI, ADD, SUB, SLT

Write, assemble and run a program on your CPU that acts as a high-level test-bench for it. This program needs to exercise all of the portions of your design and give a clear pass / fail response.

Every single module of Verilog you write needs to be **commented** and **tested**. The assembly program test only tests at a high level – each module needs to be tested on its own with a Verilog test bench.

The Reach

You must choose at least one enhancement to this assignment so that it better suits your interests.

Note: These vary in difficulty. Try to optimize for how happy you will be when you finish the lab. This includes how much time it will take and how interested you are in a particular 'reach'.

The Reach: Completionist

Implement 10 additional opcodes. You can make this easy on yourself by picking "overlapping" opcodes, such as {add, addi, addu, addiu}. This may be the easiest reach, you slacker you.

The Reach: Cool Math

This reach emphasizes assembly.

Write the assembly necessary to have your processor calculate at least two of the following functions with at least 24 bit precision: sine, tangent, arcsine, arctangent2, square root, log2, log natural, exp. Pick an IQ representation appropriate to the functions chosen.

In your report, be sure to include proof of your accuracy. Note that if you try to do this in MATLAB it will tend to do the math in double-precision, which is not what your processor will do.

Hint: Newton-Raphson, CORDIC. You may need to implement additional primitive ops.

The Reach: Make It Real

This reach emphasizes real world tools.

Put your design on an FPGA (we have Spartan 3s available). Your test program should be modified so that it reacts to external stimuli and provides some indication of success. Your final report should include a list of what resources on the FPGA you are using – how many LUTs? BRAMs? Hard Resources?

The Reach: Not a Single Cycle

This reach emphasizes logic design.

Instead of creating a single cycle design, create either a multicycle or pipelined design. Make sure that you address any hazards that this design may have; You can do so either in the hardware design or in the compiler.

The Reach: Not MIPS

This reach emphasizes alternate architectures.

You may choose to modify your processor so that it no longer a MIPS-like processor. This may be to emulate some other architecture (M0?), or to go your own way entirely. Your final report needs to be very explicit about the path you take here.

Note: This will force you to find some other way to assemble your program in to machine code! "By Hand" is an acceptable response, but be aware of what you are signing yourself up for.

The Reach: Choose Your Own Pain

If there is an enhancement that is not listed here that you think should be, do it! Clear the enhancement with Eric VanWyk by the Friday after the lab is given. This is to be done by including your own "The Reach:" text to be included in next year's version of this lab.

The Writeup

As before, each lab has a semi-professional write up associated with it. I'd like special attention paid to how your tests prove the functionality of your processor and why your reach is awesome.

The Demo

We will give demos in class, roughly 8 minutes per team, the class after this lab is due.

These demos are required regardless of how well your code does or does not work.

Hints / Notes

Assembling

MARS is a decent assembler – you can see the machine code (actual bits) when you are debugging.

Here is the encoding for "JR". It is an r-type instruction, which means that its opcode is b000000. The function field is '8' The target register that contains the new PC value is stored in rs, and rd = rt = sa = 0.

http://www.cs.sunysb.edu/~cse320/MIPS_Instruction_Coding_With_Hex.pdf

Timing

Timing is not relevant for this lab unless you choose to make it relevant.

Design Reuse

You may freely reuse code created for previous labs. In fact, you really really really ought to.

Behavioral vs Structural

Use Behavioral Verilog or Structural Verilog however you'd like. Each component you create needs to be in its own structural module, so that you can test it appropriately. If it is a box you draw on the whiteboards during SingleCycle, MultiCycle or pipelined board work, it needs its own structural module. If it is inside one of those boxes, oh behave.

Initializing Memory

The Behavioral Verilog slide deck includes a slide on how to initialize a memory (e.g. data memory or instruction memory) from a file with $readmemb or $readmemh. This will make your life very much easier!

For example, you could load a program into your data memory by putting your machine code in hexadecimal format in a file named "file.dat" and using something like this for your instruction memory.

```
module memory(clk, regWE, Addr,
                DataIn, DataOut);
  input clk, regWE;
  input[9:0] Addr;
  input[31:0] DataIn;
  output[31:0]  DataOut;

reg [31:0] mem[1023:0];
always @(posedge clk)
  if (regWE)
    mem[Addr] <= DataIn;
initial $readmemh("file.dat", mem);
```

```
assign DataOut = registers[Addr];
endmodule
```

Note: You may need to fiddle with the ‘Addr’ bus to make it fit your design!