ENGR xD52: MP b011

Due December 4th 11PM EST

The purpose of this machine problem is to specify, create, and test a 32-bit CPU that can run a specified program. This is (intentionally) the least scaffolded lab this semester.

Work in groups of 3 to 4. This may require some team shuffling. One representative from each team must submit all deliverables electronically to comparch14@gmail.com as a single compressed archive. Please format the email's Header as "[MP3] Name1 Name2 Name3". As usual, this archive will include all relevant technical files as well as a pdf write-up.

The Work Plan

This subdeliverable is due within 36 hours as an email to comparch14@gmail.com with the subject line formatted as [MP3 Work Plan] Name1 Name2 Name3".

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).

Submit a scheduling update by 11/23/14.

Provide the final scheduling analysis in your final deliverable.

The Program

Your processor will need to execute the following program:

```
int main() {
 int fib4 = Fibonacci(4);
 int fib10 = Fibonacci(10);
 return fib4+fib10;
}

int Fibonacci(int x) {
 if (x == 0) return 0;  // Stopping conditions
 if (x == 1) return 1;
 int fib_1 = Fibonacci(x - 1);
 int fib_2 = Fibonacci(x - 2);
 return fib_1+fib_2;
}
```

Compile to Assembly

Recreate the code above in assembly as faithfully as possible; do not make any optimizations. Your code will be fully recursive, and will not tail-call or trampoline.

For ease of grading, use the register allocation provided on slide 22 of deck 10.

Deliverables

1) The Fibonacci code in its own .asm file. We will be checking the following things:
    a. $v0 has the value 58 (decimal)
    b. The code is written **recursively**
    c. The stack pointer has returned to its initial value (balanced stack)

2) A diagram that shows what is pushed onto the call stack for each call to Fibonacci(x). For reference, my solution pushed 3 elements onto the stack for each call. You may need to push additional elements, it is unlikely you will need to push fewer. This should be in your write-up.

3) The list of instructions that you need to implement, based on this program. This should be in your write-up.

The Processor

Create a CPU that is able to execute the Fibonacci Test Program. To do so, you will need to first identify what instructions are necessary for this program.

Style

We are not dictating what type of CPU to make. There is some debate over whether single-cycle or multi-cycle is easier. Selecting a pipelined architecture is definitely more difficult.

If you choose an architecture other than 32-Bit MIPS you are **way** off the reservation. I will be unable to support you if you choose to shoot yourself in the foot in this manner, but I will not stop you.

Synthesizability

We will **not** be requiring you to put this on an FPGA. However, you will be putting it through the synthesizer to check what types of structures it is generating. Make sure your code is mapping correctly onto the hard logic in our FPGAs. For example, make sure that it is using Block RAMs instead of Flip Flops and LUTs to create the memory structures.

The register file will likely synthesize as 2 parallel BRAMs; A regfile has 3 ports and a BRAM has only 2. To get around this, it creates two memories with their write ports tied together.

Your existing ALU might not make use of hard adders, this is ok.

Deliverables

1) A written description and block diagram of the architecture you have chosen.

2) All relevant .v and .do files in the archive.

3) A "DoFibonacci.do" file. Running this should execute the provided program, display relevant waveforms, and end execution when the program is finished. Include a brief description of where to look in the waveforms to verify the answer in comments in this file.

4) Include the following statistics in the writeup:
    a. Resources used (Slices, BRAMs, MULT18x18s, etc)
    b. Maximum Clock Speed
    c. Total MIPS. If you need to, provide this for a specific mix of instructions of your choosing.

The Test Scaffolding

You will provide a sequence of tests that cover the entire design of your processor. It is highly recommended you develop this before / in tandem with the development of the processor.

Deliverables

1) A written overview of your test strategy. You may want to include a second copy of your block diagram with annotations as to which test(s) cover which aspects. Every aspect of your processor needs to be covered by your total test strategy.

2) For 2 of the tests, include a more detailed description. This should include enough information for someone else (a NINJA) to run them. At a minimum, this should include:
    a. Instructions on how to run the test. For example, "Run RegisterFileTest.do, verify the following…"
    b. How the test works – what does it actually do?
    c. At least 2 types of design errors the test could catch.

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

Design Reuse

You may freely reuse code created for previous labs, even if another team wrote it.

Each example of reuse should be documented.

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

Psuedo-Instructions

There are many instructions in MARS that aren't "real" instructions, but instead map onto other instructions. For example, "move $destination, $source" is shorthand for "add $destination, $source, $zero". My version of this used "li" and "move".

Behavioral vs Structural

Use Behavioral Verilog or Structural Verilog however you'd like. Note that if you turn the crank too far in the behavioral direction you will likely generate extremely inefficient / repeated structures. Keep an eye on this in the synthesis output!

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 = mem[Addr];
endmodule
```

Note: You may need to fiddle with the 'Addr' bus to make it fit your design, depending on how you handle the "address is always a multiple of 4" issue.

Note: This only works in ModelSim. It will be ignored in Xilinx ISE (this is ok).

Memory Configuration

In MARS, go to Settings -> Memory Configuration. Changing this to "Compact, Text at Address 0" will give you a decent memory layout to start with. This will put your program (text) at address 0, your data at address 0x1000, and your stack pointer will start at 0x3ffc.

Note: You will need to manually set your stack pointer in ModelSim. This is done automatically for you in MARS.