CprE 381 – Computer Organization and Assembly Level Programming Spring 2019

Lab #3

[Note: In this lab you will get to create some more components that will prove to be useful when you begin your MIPS processor design projects. At first glance it will seem even longer than Lab #2 but it should be both a shorter and more worthwhile experience. Note that this is still an individual lab assignment, even though you are required to sign up for a project team before beginning. Team learning is always encouraged, as long as the end product is intellectually your own.]

0) Prelab. Read *Free Range VHDL*, pages 51-68, 71-85. At the end of Chapter 5, answer question 5. At the end of Chapter 7, answer exercise 2. Read through the entire lab documentation, in particular through part (2).

Also, sign up for your **Project Teams** with your TA at the beginning of your lab section. Failure to do so may lead to delays in grading this lab. Please abide by the following guidelines / recommendations when selecting your project team:

- Teams should contain 2 students, or 3 if there are an odd number of students in a lab. Single student "teams" are not acceptable without prior instructor approval. Teams of 3 will have some additional work.
- Select your team from the students in your own lab section only.
- Each team member will be accountable for understanding and being able to explain their team's entire project during demos.
- Sign up for a group in the Term Project group set on Canvas (see https://community.canvaslms.com/docs/DOC-10516-421264913). The group you and your partner claim must begin with your section letter.
- In # your Lab 3 report PDF, provide the Canvas group name for your project team and a listing of its members. On a scale of 1-10, how comfortable with VHDL does each team member currently feel?

1) The MIPS **Register File** contains 32, 32-bit registers, along with two ports for reading (corresponding to `rs` and `rt` in a MIPS ISA R-type instruction), and one port for writing (corresponding to `rd` in a MIPS ISA R-type instruction). Provide your solution to this problem (VHDL code, simulation waveforms) in a folder called 'P1/'.

(a) Draw the interface description for the MIPS register file. Which ports do you think are necessary, and how wide (in bits) do they need to be? *[Some things to think about when answering this question: will this need to be a synchronous (i.e. with a clock) circuit? How would one be able to differentiate between read and write requests? If you are unsure about the answers to these questions from lectures, some answers can be found in P&H chapter 4. Verify with your TA before proceeding.]*

(b) The first component that would be needed is an individual register. A behavioral VHDL edge-triggered flip-flop (with parallel access and reset) is provided in file *dff.vhd*. Create an N-bit register using this flip-flop as your basis. *[This should now be simple if you choose to do this using structural VHDL with generics, and trivial if you copy and modify the* dff.vhd *file directly. Either way is acceptable.]*

(c) Use ModelSim to test your register design to make sure it is working as expected, and include a waveform screenshot in your report PDF. A sample testbench that also incorporates a clock generator can be found in *tb_dff.vhd*. *[You will need to modify this slightly for your N-bit version.]*

(d) A decoder is a logic structure that takes in an N-bit value, and sets one bit out of a 2^N-bit output value based on the corresponding decimal representation (e.g. a 3:8 decoder will output "00000010" when the input is "001", will output "00010000" when the input is "100", and will generally set only bit *I* when the binary input corresponds to *I* in decimal). What type of decoder would be required by the MIPS register file and why?

(e) Based on your answer to part d) implement the appropriate decoder using either structural or dataflow VHDL. Use ModelSim to test your decoder design to make sure it is working as expected. *[It is not easy (and not worth it) to make this a general N:2^N decoder. Free Range VHDL provides a dataflow representation of a decoder using* "with-select-when" *that will greatly simplify this task.]*

(f) For a given read request on the register file, we only want a single register's output value, even though all 32 registers can and will be read in parallel. A 32-bit 32:1 multiplexor can be used to accomplish this task. Based on the dataflow and structural VHDL we have already learned, there are at least five different ways to implement a 32-bit 32:1 multiplexor. In your write-up, describe and defend the design you intend on implementing for the next part.

(g) Based on your answer to part (f), implement your 32-bit 32:1 multiplexor and use ModelSim to test your design to make sure it is working as expected. *[You can design this component however you want. Generally a dataflow implementation will be easier to code and faster to simulate. On the flip side, a structural implementation provides more detail about the underlying logic layout. Hybrid approaches are also valid.]*

(h) Draw a (simplified) schematic for the MIPS register file, using the same top-level interface ports as in your solution for part a), and using only the VHDL components you have created in parts (b), (e), and (g). Keep in mind that although there are 32 32-bit registers in the MIPS register file, register `$0` is required by the ISA to return the value of zero at all times. *[To keep things simple, you do not need to draw out each of the 32 registers and corresponding control signals. Instead I recommend simplifying the structure as $0, $1, $2, …, $31.]*

(i) Based on your answer to part h) fully implement the MIPS register file and use ModelSim to test your design to make sure it is working as expected. *[This should be mostly structural code. For register* `$0`*, you can set a signal's value to all '0's using the somewhat counterintuitive* value <= (others => '0'); *VHDL assignment statement. An alternative strategy is to set the i_RST port to '1' at all times.]*

2) Given the components you have already created in this and the previous week's lab, you can now implement the core datapath of a MIPS-like processor. Provide your solution to this problem (VHDL code, simulation waveforms) in a folder called 'P2/'.

(a) Until we have a memory module, the only way to store any non-zero values in the register file is to make use of immediate-type arithmetic instructions. Add an additional 32-bit 2:1 multiplexor to the second input of the adder/subtractor design from Lab #2, using a new control signal called ALUSrc. Your design should now behave according to the following control table *[note that this is not strictly correct MIPS, which does not need or have a `subi` instruction]*:

```
nAdd_Sub     ALUSrc    |   Operation
-------------------------------------
    0           0      |    C ← A + B;
    0           1      |    C ← A + immediate;
    1           0      |    C ← A - B;
    1           1      |    C ← A - immediate;
```

(b) Draw a symbol for this MIPS-like datapath. *[Think about all of the signals you will need: control input signals, data input signals, clock, reset, etc.]*

(c) Draw a schematic of the simplified MIPS processor datapath consisting only of the component described in part (a) and the register file from problem (1).

(d) Implement this simplified MIPS-like datapath using structural VHDL. Create a VHDL testbench to demonstrate that your datapath can support the following code. Include in your report waveform screenshots that demonstrate your properly functioning design. *[You do not have to assemble these instructions into their proper MIPS machine language equivalents. Instead, determine what values inputs to your design would correspond to for these instructions. For example, for the first instruction, rs=0, rt=x, rd=1, nAddSub='0', ALUSrc='1', regWrite='1'. Each instruction should complete within a single cycle. When writing your testbench, it can be helpful to change your inputs on the non-active edge of the clock to clearly show which cycle the datapath is operating in.*]

```
addi   $1,  $0,  1        # Place "1" in $1
addi   $2,  $0,  2        # Place "2" in $2
addi   $3,  $0,  3        # Place "3" in $3
addi   $4,  $0,  4        # Place "4" in $4
addi   $5,  $0,  5        # Place "5" in $5
addi   $6,  $0,  6        # Place "6" in $6
addi   $7,  $0,  7        # Place "7" in $7
addi   $8,  $0,  8        # Place "8" in $8
addi   $9,  $0,  9        # Place "9" in $9
addi   $10, $0,  10       # Place "10" in $10
add    $11, $1,  $2       # $11 = $1 + $2
sub    $12, $11, $3       # $12 = $11 - $3
add    $13, $12, $4       # $13 = $12 + $4
```

```
sub   $14, $13, $5       # $14 = $13 - $5
add   $15, $14, $6       # $15 = $14 + $6
sub   $16, $15, $7       # $16 = $15 - $7
add   $17, $16, $8       # $17 = $16 + $8
sub   $18, $17, $9       # $18 = $17 - $9
add   $19, $18, $10      # $19 = $18 + $10
addi  $20, $0,  35       # Place "35" in $20
add   $21, $19, $20      # $21 = $19 + $20
```

Credit: This lab was originally created by Dr. Joe Zambreno.