Instructions

Objective

In this lab, you will develop a MIPS register file, which is a 32 address memory of 32 bit words that will be used to store register values in your MIPS CPU. To interact with your register file, you will also implement a simple reverse-polish-notation (RPN) calculator, using your existing ALU to provide math operations. You will also construct a suite of tests to ensure that your design performs as intended.

Starting with this and all future labs, rather than a specific list of step-by-step instructions, you will instead be provided with a set of design requirements describing the code you must implement. Beginning with this lab, you will have much greater creative freedom with respect to how your project is organized and implemented.

Deliverables

- Your assignment directory, packed using the provided script.
- **You will be turning in two separate files, one `.7z` generated by the provided script, and one PDF containing your report**.
- Each group only needs to submit once via either partner.

Design Requirements

Your design must use the included `regfile` and `rpncalc` modules with the module lines unchanged. These modules will be instantiated to test your design. You will not be awarded points if these module lines are changed or renamed. You will need to instantiate them as needed in your top level module or elsewhere to complete your design. You may create as many other modules nested below these as you wish however. A top-level-module is provided for use with the board, in which you can instantiate your `rpncalc` module. You should carry over your 7 segment decoders from the previous lab. You may also re-use the ALU from the previous lab to carry out math operations.

Register file requirements:

- (i) The register file must have two 32 bit read ports and associated 5 bit read addresses ($\lceil \log_2(32) \rceil = 5$).
- (ii) The register file must have one 32 bit write port and associated 5 bit write address.
- (iii) On any rising clock edge, if the write enable signal is asserted, the value of the write data signal must be written to the memory address specified by the write address.

- (iv) Both read data ports should always have the value stored at the memory address defined by the corresponding read address, except as noted below.
  - (iv.a) If the write enable signal is asserted, then a read via either port of the address defined by writedata should instead yield the value of write data. In other words, both read ports must implement write-bypassing.
  - (iv.b) Any read to the address zero will yield a value of zero via the relevant read data port.

RPN calculator requirements:

- (v) The RPN calculator should feature a 31 address stack, implemented using your register file.
- (vi) The RPN calculator should always show the value at the top address of its stack in hexadecimal on the rightmost 4 hex displays (`HEX3`, `HEX2`, `HEX1`, `HEX0`).
- (vii) The RPN calculator should always show the value at the second-to-top address of its stack in hexadecimal on the leftmost 4 hex displays (`HEX7`, `HEX6`, `HEX5`, `HEX4`).
- (viii) The RPN calculator will read values from the rightmost 16 switches (`SW15` to `SW0`) zero-padded to be 32 bits long, and a 2 bit mode from the leftmost two switches (`SW17` and `SW16`).
- (ix) When any `KEY` is depressed, the RPN calculator will perform one of 16 possible operations depending on the `KEY` and the current mode. The operations are defined in Appendix I.
- (x) The 8 green LEDs (`LEDG7`, . . . `LEDG0`) should be used to display a counter, in binary, of the number of elements currently on the sack (with `LEDG7` being the most significant bit, and `LEDG0` being the least significant bit).

General requirements:

- (xi) A specific testing strategy must be **defined** and **implemented** to ensure that the implemented code performs as it should. This might take the form of a Verilog test bench, a ModelSim TCL script, or some other method. Any test scripts or code must execute on the Swearingen Linux lab computers. A `README` file (or similar documentation) describing how to execute the test suite should be provided. The testing code may assumed that a DE2-115 will be connected to the computer it is running on and may be programmed if needed.
- (xii) You must connect and utilize the connections in the `rpncalc` and `regfile` modules as described by their comments and this document.

Report

You must write a report. The report should be written using a reasonable choice of font and other stylistic options. The report **must** be in PDF format.

The report must include the following information:

- Three to five paragraphs describing your testing strategy (rationale, design decisions, etc.). Imagine the reader as a new person having just joined your team who will need to use your tests.
- A justification of how the test cases you have written demonstrate and prove the correctness of the design under test. The precise nature of this justification will vary depending on your approach and style. It may include text, figures, tables, code listings, etc. Roughly 1-3 (but no more than 5) pages is expected.
  - Some examples of suitable justifications would include (but are not limited to):
    * A short (~ 1 sentence) description of each test case in a suite of test cases (which might be implented using non-synthesizeable Verilog in ModelSim, as a ModelSim TCL script, by instrumenting SignalTap, or by some other method).
    * Annotated screenshots of ModelSim waveforms which demonstrate correct operation.
    * Annotated screenshots of SignalTap waveforms or memory monitors which demonstrate correct operation.
    * Commented source code for a software simulator, and a script which generates inputs to ModelSim and checks outputs against the simulator's results.
  - Examples of **insufficent** justifications would include (but are not limited to):
    * A description of a fully manual test procedure.
    * A video or photographs of a fully manual test procedure.
    * A collection of test cases which leaves significant portions of the design untested.
    * A collection of test cases or vectors without any description.
- A short analysis (1-3 paragraphs) of your implemented design using your test methodology. You should indicate if your design works, and if not why it does not.

The specific organization, layout, prosaic style, etc. of your report is left up to your discretion and good judgment.

Rubric

- (A) 20 points – correctly implemented register file per the supplied requirements (*(i)* . . . *(iv)*).
  - Partial credit may be awarded on a case-by-case basis for designs that do not work but demostrate partial understanding.
- (B) 30 points – correctly implemented RPN calculator per the supplied requirements (*(v)* . . . *(x)*).

  - (B.1) 15 points – push, pop, and swap (`00,3`, `00,2`, `11,3`) operate correctly (5 points each) (req. *(ix)*).
  - (B.2) 10 points – ALU operations operate correctly (1 point per op) (req *(ix)*).
  - (B.3) 5 points – stack usage counter on green LEDs (req *(x)*).

- (C) 40 points – report.
  - (C.1) 10 points – description of testing strategy.
  - (C.2) 20 points – justification of tests.
    * (C.2.a) 10 points – clear description of specific tests or test cases implemented.
    * (C.2.b) 10 points – thoroughness of test cases (partial credit awarded for tests that are partially thorough, proportional to thoroughness).
    * **NOTE** it is the author's task to convince the reader that the tests are complete, accurate, and effective.
  - (C.3) 10 points – analysis of implemented design.

- (D) 10 points – pre-lab assignment.
  - Points are awarded based on number of correctly completed problems, to be submitted via Moodle/Dropbox. See `prelab.pdf`.

Maximum score: 100 points.

Additionally, the following may cause you to lose points:

- If the code provided in your submission **does not match** what is shown in your report (i.e. in screenshots, code listings, etc.), you will be given a failing grade on the assignment.
  - This includes functional descriptions of your code. For example, if you did not implement any of the ALU operations, but your report indicated you did, you will receive a failing grade.
- **If your report is submitted without your project code (or your project code without your report), you will be given a failing grade on the assignment.**
- Your code may be run through Moss, and your report through plagiarism detection software such as TurnItIn. **Plagiarism and other forms of cheating will be reported to the academic honesty department, which may result in a grade penalty.**
- Failure to follow the styling guidelines will result in a one letter grade penalty **($\frac{1}{10}$ of maximum points for the assignment)**.
  - The same styling requirements apply as for the ALU lab.
- Due to the nature of the design to be implemented, this lab cannot be graded in an "error carries forward fashion". For example, a regfile implementation that does not work will very likely result in an RPN calculator that also does not work.

  - Your design will be compiled with your register file, ALU, etc. In the event that, for example, your register file is not working properly, but your RPN calculator works around this and still exhibits the correct behavior, you would still get full credit for the RPN calculator portion of the assignment.
  - To verify functionality, your `regfile` and `rpncalc` modules may be instanced via an alternate top-level module and compiled under ModelSim or Quartus. **This is how points *(A)* and *(B)* in the ruberic above will ba assessed**.

- Failure to follow requirement *(xii)* will make it impossible to score your design for sections *(A)* and *(B)* of the rubric, and you will receive a score of 0 points for those sections.

Hints & Tips

- Write enable is active-high.
- There are several ways to approach requirement *(iv.b)* that are correct.
- Your read data ports should obtain their values from combinational logic (i.e. an `always_comb` block)
- The term **memory port** generally implies one or more signals used to interact with a memory in some way. In general, a read port has a read address and a read data signal, and a write port a write enable, write data, and write address signal.
- Your stack may "start" at either the "upper" or "lower" end of your register file. Just be sure to keep which direction you are going straight. However, it is usually easier to keep straight if you start the stack at `0x01` and grow "up".
- You are encouraged to use a state machine to keep keep track of the RPN calculator's operation.
- The behavior if more than 31 values are pushed onto the stack is undefined and will not be tested (you don't need to test it either).
- Your test approach should be automated, no manual testing strategy will be sufficiently rigorous to ensure that all aspects of the design are working properly. One approach might be to write a test case for each item in the list of requirements. Another approach might be to consider every possible branch in your code and write an appropriate test based on thereon.

  - It's ok to have *some* level of manual interaction for running your test suite. For example, having to execute several separate scripts. It should require not require more than 1-2 minutes of total manual human interaction per complete run of all test cases (it may take longer for the automated portions to execute however).
  - Although it is desirable for a test suite to output a single final pass/fail after it runs, it's also ok if the results require manual interpretation. Be sure to clearly describe how to interpret test results in your report.

- In the "short" column in the table from Appendix I, the left part of each pair of numbers is the binary value of `SW17` and `SW16`, and the right part is the decimal number of the `KEY`. For example, `01,1` would indicate that `SW17` is in the `0` position, `SW16` is in the `1` position, and `KEY1` is being depressed. This is simply a convenient notational convention you may elect to use if you wish.
- While you are permitted to use any strategy you would like to implement your test suite, the suggest approach is to start with the test bench used in the ALU lab as a starting point to write a test bench and test vectors for your design.
  - Remember that your RPN calculator design may take several clock cycles to respond to a key press. You may need to wait several cycles between an input and the corresponding output.
  - It might be useful to have a separate test bench and set of test vectors for the register file and the RPN calculator.
  - A useful technique for checking the results of your RPN calculator via your test bench might be to build a "hex decoder" module in your test bech, which converts 7 bit hex display outputs back to their 4 bit inputs.
  - Unlike the ALU lab, this lab requires a clock signal to be generated. Refer to the lecture slides for examples.
- You do not need to perform de-bouncing for the `KEY` inputs.
- You may assume that only one `KEY` is depressed at a time, although handling and testing for such cases is a good learning exercise which you are encouraged (but not required) to do anyway.
- You are encouraged to lean on tools you are already familiar with. You might wish to build a calculator, simulator, etc. in your favorite programming language. Please indicate if you have done so in your lab report though.
  - Within reason, you may utilize auxiliary or supporting code from the internet or other sources. For example, an input tokenizer or parser to read data into a simulator might be sourced from a textbook or web page. If you choose to use such code, you **must** cite it and provide proper attribution to it's origonal author; failure to do so constitutes plagarism.
  - You may **not** use code from sources other than yourself and your lab partner to implement core functional aspects of your design, such as your register file or RPN calculator modules, your test cases, test bench and so on.
- A `simulate` command has been provided to use with `csce611.sh` as in the ALU lab. This script will attempt to run a simulation on a module named `CSCE611_regfile_testbench`. You are not required to use this command if you do not wish to do so, however you will need to define the noted module if you do.
- The rationale behind requirements *(vi)* and *(vii)* is that the top two values must be known in order to execute any two-operand operators (add, sub, mult, etc.).

- In principle, it would be possible to only have access to the top value, and pop it off into an accumulator register before executing a two operand computation, but this would add additional delay and complexity.

Appendix I - Table of RPN Calculator Operations

Short	SW17	SW16	KEY	Description
00,3	0	0	3	push the current switch value onto the stack
00,2	0	0	2	pop the topmost value from the stack
00,1	0	0	1	pop the topmost 2 values from the stack, and push their sum onto the stack (ALU op 0100).
00,0	0	0	0	pop the topmost 2 values from the stack and push their difference onto the stack (ALU op 0101) (second to topmost value less topmost value).
01,3	0	1	3	pop the topmost 2 values from the stack and push the lower 32 bits of their unsigned product onto the stack (ALU op 0111).
01,2	0	1	2	pop the topmost 2 values from the stack and push the second to topmost value shifted left by the topmost value onto the stack (ALU op 1000).
01,1	0	1	1	pop the topmost 2 values from the stack and push the second to topmost value shifted right (logical) by the topmost value onto the stack (ALU op 1001).
01,0	0	1	0	pop the topmost 2 values from the stack, if the topmost value is less than the second to topmost, push 1 onto the stack, and otherwise push 0 onto the stack (ALU op 1100)
10,3	1	0	3	pop the topmost two values from the stack, and push the bitwise AND of these values onto the stack (ALU op 0000).
10,2	1	0	2	pop the topmost two values from the stack and push the bitwise OR of these values onto the stack (ALU op 0001)
10,1	1	0	1	pop the topmost two values from the stack and push the bitwise NOR of these values onto the stack (ALU op 0010).
10,0	1	0	0	pop the top two values from the stack and push the bitwise XOR of these values onto the stack (ALU op 0011).
11,3	1	1	3	pop the topmost two values from the stack and push them back onto the stack in reverse order.
11,2	1	1	2	unused – you may implement your own operation if you wish.
11,1	1	1	1	unused – you may implement your own operation if you wish.
11,0	1	1	0	unused – you may implement your own operation if you wish.

