Instructions

Objective

In this lab, you will augment the existing MIPS CPU which you have implemented for the R and I type lab. Your CPU will be expanded to accommodate jump and branch instructions. To demonstrate the operation of these new instructions, you will also implement a MIPS assembler program which performs a square root operation on a number entered via the switches.In addition, you will define and implement a testing methodology that

demonstrates the soundness of your design using test benches or any other tool of your liking to ensure that all functionality is correctly implemented. Finally, you will write a report describing this test methodology, and the results thereof.

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 `cpu` module with the module lines unchanged[1]. This module will be instantiated to test your design. Your square root program will also be tested by executing it within your CPU. See *Appendix I* for a listing of J type instructions.

J-Type instruction requirements:

- (i) The instructions beq, bne, bgez, j, jal, and jr must be implemented as described in *Appendix I*

Assembler Program Requirements:

- (ii) Your MIPS assembler program should be stored in a file named `sqrt.asm`, which should be a MARS-compatible MIPS program. It must read a value from GPIO, and output a decimal version of the square root of this value via GPIO.

[1]You may change `output` lines to `output logic` or vice-versa at your discretion, so long as the signal names and widths are unmodified.

- The value should be shown in fixed point, with 3 fractional and 5 non-fractional decimal digets. Thus, when displaying the value 1.41421, the digits `0x00141421` should be the value to the GPIO output, and hence to the hex displays.
- Your program should compute a result with a precision of not less than $\frac{1}{100000}$.

General requirements:

- (iii) When `KEY0` is pressed, your CPU should reset itself (i.e. this should be your `rst` signal).
- (iv) The CPU requirements for the R/I type lab still hold, in that your implemented design should still satisfy all CPU-related requirements of the R/I lab, in addition to the new requirements.
  - In particular, it is important that your GPIO continues to function identically.
  - Your instruction memory must also still be named `instmem.dat`.
- (v) 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.
- (vi) You must connect and utilize the connections in the `cpu` module as described by their comments and this document.
- (vii) You should implement a top-level module in the `CSCE611_lab_ri.sv` file which instantiates your CPU, connects the switches (`SW`) to the GPIO input, and the HEX displays (`HEX0` … `HEX7`) to the GPIO output.

Suggested Approach

As before, you may deviate from this suggested approach. It is intended to give you a direction and general strategy of how to complete your design. If you want to try something else, feel free, so long as your design meets the listed requirements.

You should start by copying over your files (ALU, register file, CPU, control unit, etc.) from the R/I type lab. This lab is intended to build directly upon the R/I type lab, and you will keep nearly all the code written for that lab.

This lab should require no major architectural changes to your CPU or test bench, and will mostly consist of adding a few new muxes…

To begin, the following is an overview of signals you may wish to keep in mind when designing your changes. The rest of this section will refer to these signals as described below:

- `pcsrc_EX`, a control signal which determines how the next value for the `PC` register is calculated
- `reg_addr_EX`, the read value for jr instructions
- `branch_addr_EX`, the computed branch target for branch instructions
- `jtype_addr_EX`, the immediate jump target for J-type instructions
- `stall_EX`, a control signal which indicates the control unit should ignore the current `instruction_EX`, obtained by delaying `stall_FETCH` one cycle.
- `stall_FETCH`, a control signal asserted when the fetch stage detects an unconditional jump.

`PC` will now be updated by a mux controlled by the `pcsrc_EX` signal. The inputs to this mux will be:

- `PC_F + 1` (this is how `PC` is updated in the R/I lab)
- `branch_addr_EX`
- `jtype_addr_EX`
- `reg_Addr_EX`

`stall_FETCH` will need to be driven by decoding the first 6 bits of `instruction_FETCH`, and checking if they match the opcode for `jal` or `j`, indicating that a stall is needed.

The `rdrt_EX` mux should be expanded to have a third input, `5'd31`, which allows `regdest_EX` to take the value 31 when `jal` instructions are executed.

Your control unit will need to be expanded to generate appropriate values for the `stall_EX` and `pcsrc_EX` signals. Be mindful when updating `rdrt_EX` to support it's new input, as the width of the control line will have expanded from 1 bit to 2.

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 – `sqrt.asm` program.
- (B) 15 points – jump instructions (j, jr, jal), 5 pts each.
- (C) 15 points – branch instructions (beq, bne, bgez), 5 pts each.
- (D) 40 points – lab report.
  - (D.1) 10 points – description of testing strategy.
  - (D.2) justification of tests.
    * (D.2.a) 10 points – clear description of specific tests or test cases implemented.
    * (D.2.b) 10 points – thoroughness of test cases (partial credit awarded for tests that are partially thorough, proportional to throughghness)
  - (D.3) 10 points – analysis of implemented design.
- (E) 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.
- As in the previous lab, it will be impossible to evaluate rubric items *(A)*, *(B)*, and *(C)* if GPIO is not implemented, your cpu does not load it's instruction memory from `instmem.dat`, and/or if your `cpu.sv` has incorrect or unused inputs. This will result in a score of 0 on those items.

Appendix I - MIPS R-Type Instructions

NOTE: *labels* (as used in branch instructions) are translated to a 16-bit *offset* by the assembler.

j *target* :

Unconditionally jump to instruction at *target*.

31-26	25-0
000010	*target*

jal *target* :

Save address of the next instruction in register 31 and then unconditionally jump to instruction at Target.

31-26	25-0
000011	*target*

jr ***rs*** **:**

Unconditionally jump to instruction at address stored in *rs*.

31-26	25-21	20-6	5-0
000000	*rs*	don't care	001000

beq ***rs*****,** ***rt*****,** ***label*** **:**

If the integer in register *rs* is equal to the integer in register *rt*, then increment the program counter (PC) by *offset*.

31-26	25-21	20-16	15-0
000100	*rs*	*rt*	*offset*

bne ***rs*****,** ***rt*****,** ***label*** **:**

If the integer in register *rs* is not equal to the integer in register *rt*, then increment the program counter (PC) by *offset*.

31-26	25-21	20-16	15-0
000101	*rs*	*rt*	*offset*

bgez ***rs*****,** ***rt*****,** ***label*** **:**

If the integer in register *rs* is greater than or equal to zero, then increment the PC by *offset* *.

31-26	25-21	20-16	15-0
000001	*rs*	00001	*offset*