Project Checkpoint 2

Full ALU

Logistics

This is the second Project Checkpoint for our processor. We will post clarifications, updates, etc. on Canvas and Ed.

- Due: **Thursday, September 26, 2024,** by **11:59 PM** **(Duke time)**
    - Late policy can be found on the course webpage/syllabus

Introduction

Design and simulate an ALU using Verilog. You must support:

- a **non-RCA** adder with support for addition & subtraction (that you have done in the last checkpoint)
- bitwise AND, OR **without** the built-in &, &&, |, and || operators
- 32-bit barrel shifter with SLL (Logical Left Shift) and SRA (Arithmetic Right Shift) **without** the <<, <<<, >>, and >>> operators

Module Interface

Designs which do not adhere to the following specification will incur significant penalties.

Your module must use the following interface (n.b. It is the template provided to you in alu.v):

```
module alu(data_operandA, data_operandB, ctrl_ALUopcode,
ctrl_shiftamt, data_result, isNotEqual, isLessThan, overflow);

   input [31:0] data_operandA, data_operandB;
   input [4:0] ctrl_ALUopcode, ctrl_shiftamt;

   output [31:0] data_result;
   output isNotEqual, isLessThan, overflow;

endmodule
```

Same as PC1, this shows you the port list that your module must have along with their directions (input/output). This is required as the autograder needs correct port names & directions to test your design.

Each operation should be associated with the following ALU opcodes:

Operation	ALU Opcode	Description
ADD	00000	Performs `data_operandA + data_operandB`
SUBTRACT	00001	Performs `data_operandA - data_operandB`
AND	00010	Performs (bitwise) `data_operandA & data_operandB`
OR	00011	Performs (bitwise) `data_operandA | data_operandB`
SLL	00100	Logical left-shift on `data_operandA`
SRA	00101	Arithmetic right-shift on `data_operandA`

Since we already graded you for ADD and SUBTRACT in PC1, you won't be graded for those in this checkpoint. However, your design should still properly support those instructions along with the new ones that are required for this checkpoint. This checkpoint will only grade you for AND, OR, SLL, and SRA.

Control Signals (In)

- `ctrl_shiftamt`
  - Shift amount for SLL and SRA operations
  - Only needs to be used in SLL and SRA operations

Information Signals (Out)

- `isNotEqual`
  - Asserts true **iff** `data_operandA` and `data_operandB` are not equal
  - Only needs to be correct after a SUBTRACT operation
- `isLessThan`
  - Asserts true **iff** `data_operandA` is **strictly** less than `data_operandB`
  - Only needs to be correct after a SUBTRACT operation
- `overflow`
  - Asserts true **iff** there is an overflow in ADD or SUBTRACT
  - Only needs to be correct after an ADD or SUBTRACT operation

Permitted and Banned Verilog

Designs which do not adhere to the following specifications will receive a zero.

You can use

- Ternary assign: `assign out = cond ? high : low;` (cond, high, low must be wire(s) (or input/output ports), you should not write an expression in cond)
    - For example, assign data_result = (ctrl_ALUopcode == 2'b00000) ? Add_result : Sub_result; You cannot use '==' here, because you should not write an expression in cond.
- Primitive instantiation: `and (out, in1, in2)`
- **Bitwise not (~)**
- Generate Blocks: generate if, generate for, genvar (This is a tutorial if you do not know them: https://fpgatutorial.com/verilog-generate/)
- RCAs to construct your 32-bit adder, as long as the 32-bit adder is not RCA.
- Parameters: `parameter a=0; localparam b = a*2;`
- Any expression you like **inside the range specifier**: `a[i*15+36/2-13%2]`

You cannot use

- Behavioral Description Structures: `if ... else ... for...` (This is a loop, not the generate for, which is allowed.) `case ...`
- Megafunctions outside the range, generate control or parameter expressions: `+, -, *, /, %, **, ==, >=, <=, &&, ||, !, <<, <<<, >>, >>>`
- SystemVerilog

except in constructing your DFFE (i.e., you can use any Verilog you need to construct a DFFE).

Grading Breakdown

Test	Points	Scoring Method
Less Than	10	Proportional (-0.2 pts / test case fail until zero)
Not Equal	10	Proportional (-0.2 pts / test case fail until zero)
Or	20	Proportional (-0.2 pts / test case fail until zero)
And	20	Proportional (-0.2 pts / test case fail until zero)
Logical Left Shift	20	Proportional (-0.2 pts / test case fail until zero)

Arithmetic Right Shift	20	Proportional (-0.2 pts / test case fail until zero)

Other Specifications

Designs which do not adhere to the following specifications will incur significant penalties.

Your design must operate correctly with a 50 MHz clock. Also, please remember that we are ultimately deploying these modules on our FPGAs. Therefore, when setting up your project in Quartus, be sure to pick the correct device.

Submission Instructions

Designs that do not adhere to the following specifications will incur significant penalties.

Writing Code

- Keep all of your source files in the top-level directory.
- Make sure you structure your codes so that alu.v is the top-level entity and it contains the provided alu interface.
- Change how your repo is configured at your own risk.
- You can choose to use the GitHub repository to manage your codebase if you are familiar with that. A few suggestions:
    - Branch off of main to implement your projects and merge changes back into main when you've completed a feature or you want to test.
    - Be sure to only put files into version control that are source files (*.v).
    - Modify .gitignore at your own risk.

Submission Requirements

- When using **Gradescope** to submit your design, please submit **one .zip file** and the file should include your code and a README.md file. For **Github** submission, click 'connect GitHub', link your account, and select the correct repo and branch you want to submit.
- The submitted codes should contain **all necessary *.v modules** to execute your alu. The autograder will read and examine all .v files in the .zip file; therefore, you may be able to include subfolders but you should be aware that if you submit unnecessary .v files it could cause compile errors.
- **Please do not include testbench files in your submission.**
- A README.md (written in markdown, Github flavor) should include
    - Your name and netID,
    - A text description of your design implementation (e.g., "I used X,Y,Z to ..."),
    - A brief one-sentence functionality description of each self-designed module.
    - If there are bugs or issues, descriptions of what they are and what you think caused them.
    - Reminder: Basically, it is a general description of your design and does not exceed 1 page. However, descriptions that are too simple (e.g., contain only a few keywords) will receive a grade deduction.

Resources

Test cases in this project have already been included in your alu_tb.v in Project 1, you can simply comment back the related statements. However, the testbench used for grading will be more extensive than the one presented here. **Passing the included testbench does not ensure that you will pass the grading testbench.**

Testing your Design

We are not grading you for ADD and SUBTRACT operation for this checkpoint but it is a good idea to write tests for those in your testbench. This will ensure that your changes did not break any working code.
In the real world, verification is much more important and time-consuming than design, just like debugging typically consumes more time than writing the program itself. Here are some tips for testing your design.

- **Check the transcript first.** Make sure you address the error and understand what each warning means. There is a high possibility that the warning already tells you where the issue is.
- **Always test incrementally.** Make sure each component is correct before testing the overall design. Or it will be very hard to locate the issue.
- **Be clear about the expected value.** Do not test aimlessly and think it works as long as there is some output in the wave panel. You need to be very clear about which signal to check and what the expected value should be. If not, make sure you understand how the processor and the assembly program should work.
- **Resolve 'X' or 'Z'.** There should not be any 'x's or 'z's after reset. If there's any, you may trace the signal to find the cause. Tracing means checking its driver(s) recursively until you find all inputs are not 'x's or 'z's. If the signal is a register, check if you have initialized it during reset. When you assign a net with a smaller bit-width to a net with a larger bit-width, make sure you fill the upper bits with 0s, or it will generate 'z's.