Lab 5 - A 16-bit FPU

EE4449 - HCMUT

Objective and Overview

Design and implement a 16-bit floating point unit (FPU) compliant with IEEE 754 standard. It should perform basic arithmetic operations on 16-bit floating point operands, rounding results to the nearest value.

This is an **group** project, to be done on your Altera DE10 board.

Note: Do **NOT** use FPU in IP CATALOG for this lab.

Schedule and Scoring

If you have not uploaded anything by the dropdead date, we will assume you are no longer in the course. Why? Because the syllabus says you must attempt every project. Not uploading anything and not showing up to explain what you've done is not attempting the project — see the syllabus for details.

A Note about Collaboration

Project 5 is to be accomplished in groups. All work must be from your own team member.

Hints from others can be of great help, both to the hinter and the hintee. Thus, discussions and hints about the assignment are encouraged. However, the project must be coded and written up in groups (you may not show, nor view, any source code from other students groups). We may use automated tools to detect copying.

Use Emails/BKEL to ask questions or come visit us (203B3) during office hours.
We want to see you succeed, but you have to ask for help.

Project Overview

This project is **Simulation** only, however, your synthesis result on Quartus for the DE10 Kit still affects your credit.

Functional Specifications

The FPU must implement:

- Addition
- Subtraction
- Multiplication
- Division

For any pair of 16-bit IEEE 754 formatted inputs, it should perform the specified operation, round the result to the nearest value, and output a 16-bit IEEE 754 formatted result.

Interface (**fpu_top.sv**)

- Two 16-bit input ports for operands
- One 16-bit output port for result

Input signals to specify:

- Operation (add, subtract, multiply, divide)
- Output signals for exception flags

Implementation Guidelines

- Decode inputs into sign, exponent, mantissa
- Align and normalize mantissas
- Perform arithmetic operation on mantissas
- Normalize result mantissa
- Round mantissa to nearest value by:
    - Truncating LSB if 0
    - Adding 1 and truncating if LSB is 1
- Check for exceptions
- Encode result back into IEEE 754 format
- Testing
- Verify with testbenches, some assertions

Addition/Subtraction Algorithm (Suggestion)

- Decode the IEEE 754 formatted inputs into sign, exponent, and mantissa fields.
- Shift the mantissas to align them based on the exponent difference.
- The mantissa with the smaller exponent needs to be shifted right by the difference in exponents.
- For addition, add the aligned mantissas. For subtraction, subtract the smaller mantissa from the larger mantissa.
- Normalize the result mantissa by left shifting until the MSB is 1.
- Adjust the exponent accordingly.
- Round the mantissa to the nearest value by adding 1 to the LSB if it is 1. Truncate the mantissa if LSB is 0.
- Check for overflow or underflow based on the exponent range. Also check for inexact rounding.
- Encode the sign, updated exponent, and rounded mantissa into the IEEE 754 output format.

Key points:

- Align mantissas before adding/subtracting
- Normalize after operation
- Round mantissa to nearest
- Adjust exponent if overflow/underflow
- Encode result back to IEEE 754 format

Multiplication Algorithm (Suggestion)

- Decode the IEEE 754 formatted inputs into sign, exponent, and mantissa.
- Multiply the mantissas to generate a product.
- Add the exponents of the two inputs to get the exponent for the result.
- Normalize the product mantissa by left shifting until the MSB is 1. Adjust the exponent accordingly.
- Round the mantissa to the nearest value by adding 1 to the LSB if it is 1. Truncate if LSB is 0.
- Check for overflow or underflow and set exception flags.
- Encode the sign, exponent, and rounded mantissa into the 16-bit IEEE 754 output format.

Key points:

- Multiply mantissas
- Add exponents
- Normalize and round product
- Adjust exponent for over/underflow
- Encode result as IEEE 754

Note: The mantissa multiplication can be done with a hardware multiplier.

Division Algorithm (Radix-2)

- Decode inputs into sign, exponent, mantissa.
- Align mantissas by shifting divisor right until its MSB matches dividend's MSB.
- Perform radix-2 restoration division:
- Initialize quotient (Q) and remainder (R) to 0
- For i = 0 to n-1 (n is mantissa width):
    - Shift R left 1 bit, set LSB to dividend's next bit
    - If R >= divisor:
        - Set Q[i] = 1
        - R = R - divisor
    - Else:
        - Set Q[i] = 0
- Return Q and R
- Normalize quotient by shifting left until MSB is 1.
- Round quotient mantissa to nearest value.
- Check for underflow and inexact exceptions.
- Encode sign, exponent, mantissa into IEEE 754 format.

The radix-2 restoration algorithm performs bitwise division by subtracting the divisor and shifting until quotient is computed. This avoids doing full hardware division.

For Credit

Project 5 is entirely simulation. You should write a careful testbench for it.

Some Other Things you Should Learn

- IEEE 754 floating point format
- Overview of sign, exponent, mantissa
- Special values (infinity, NaN, denormals)
- Normalization and denormalization of floating point values
- Handling underflow and overflow conditions
- Different rounding modes and how they work
- Exception handling
    - Invalid operation
    - Divide by zero
    - Inexact rounding
- Tradeoffs between floating point precision and range
- Using fixed point vs floating point representations
- Testing floating point designs
- Generating directed tests
- Constrained random testcases
- Verification strategies
- Floating point optimizations
    - Pipelining
    - Parallelization
    - Hardware optimizations like FMA Applications of floating point units
    - Scientific computing
    - 3D graphics
    - Machine learning

How To Turn In Your Solution

This semester we will be using BKeL, simply submit the zip file with your reports and codes.

Demos and Late Penalty

We will have demo times outside of class times on or near the due date. Since we will demo from the files in your zip, it is possible that you'll demo on a following day.

Define Late: Lateness is determined by the file dates of your submission on BKeL.