ELEC6236 Digital System Design Assignment 2021/22

Prof Mark Zwolinski

November 3, 2021

1 Objective

The objective of this assignment is to design a multiplier for two 32-bit floating point numbers. Your design should be written in SystemVerilog and must simulate and synthesise correctly. Your code will be assessed automatically – by compiling your SystemVerilog code and running it through the design tools. Half of the marks for this assignment (10) will be awarded for passing tests. You must also complete a short report, describing what you have done. The other half of the marks (10) will be for this report.

This assignment is designed to test your ability to write correct SystemVerilog code and to use the design tools. The assignment is worth 20% of the module marks and should therefore take you about 30 hours to complete.

You may discuss the assignment with other students, but the code that you submit for assessment must be your own work. Please ensure you are familiar with the rules on academic integrity and the penalties for violating those rules.

2 Specification

2.1 Floating-point format

IEEE Standard 754 defines the format of floating point numbers. For this assignment, numbers should be represented in the 32-bit form (also known as binary32):

Bit 31 – Sign bit, 0 for positive numbers, 1 for negative numbers.

Bits 30-23 (8 bits) – Signed exponent. This is a power of 2, biased by 127. For example, the pattern 01111100_2 is 124_{10}. This represents the value 124-127=-3, so the mantissa (see below) is multiplied by 2^{-3}.

Bits 22-0 (23 bits) – Mantissa. This is the fractional part of the number. There is an assumed leading 1, which is not stored.

Example (from Wikipedia): 0.15625_{10} is represented in binary as 0.00101_2 (that is, 1/8 + 1/32). This can be written as $1.01_2 \times 2^{-3}$. As we have seen, the exponent, -3 is stored as $127 - 3 = 124_{10} = 01111100_2$. The mantissa is stored as $01000000000000000000000_2$ – the leading 1 is not stored, it is assumed. The sign bit is 0.

2.2 Multiplier

You are asked to design a multiplier that takes two normalized 32-bit floating point numbers and multiplies them to give a normalized result. Your multiplier should handle the 'special' numbers: zero, positive and negative infinity and 'Not a Number (NaN)'. You do not have to consider other special cases. You may use either 'round to zero' (truncation) or 'round to nearest'. Although it is possible, in principle, to design this as a combinational circuit, you must design a sequential circuit. There is no upper limit to the number of cycles that may be taken, so long as the design meets the timing requirements given in the next section.

Hints: To multiply two floating point numbers, it is necessary to multiply the mantissas and to add the exponents. So, to do this, the fields of each floating point number (sign, mantissa, exponent) need to be extracted, the implicit leading 1 of each mantissa needs to be restored. The decimal point is then assumed to be to the right of that leading 1.

The product may not be a valid floating point number because there could be more than one bit to the left of the assumed decimal point. This form of a floating point number is known as a *denormalized* number. If the mantissa of a number is shifted left or right, the exponent for that number is decremented or incremented by one, respectively. So the mantissa needs to be shifted until it is correct and the exponent is adjusted at the same time.

Although denormalized numbers are allowed under some circumstances, in this case returning a denormalized number would be wrong (why?).

2.3 Timing

Your design must be synchronous. The system must be sensitive to the positive edge of the clock (only). You should include an active low asynchronous reset that will set the contents of all registers to 0.

Your multiplier must have a 'ready' output. This should be asserted for one clock cycle only at the same time as the product is available at the output. The multiplier has one 32-bit input bus and one 32-bit output bus. When the 'ready' signal is asserted, the multiplier expects the first input on the next clock cycle and the second input on the one after that. A new calculation can begin as soon as the previous calculation is complete.

3 Assessment

Your design will be simulated with a testbench that will use some test cases. First, the reset will be tested. Then some 'corner cases' will be tried. For example:

- $0.0 * 0.0 = 0.0$
- $1.0 * 1.0 = 1.0$
- $-1.0 * 1.0 = -1.0$

The multiplier will then be tested with some arbitrary numbers. Your design will be scored for each test that is passed. The product output must be correctly normalized – you will get no marks for a result that does not conform to the IEEE standard.

The design will then be validated with Verilator and Quartus. Marks will be deducted for warnings or errors that would cause the design to be synthesised incorrectly.

4 Submission

You must submit *two* files. The first file, fpmultiplier.sv, contains your code. This may contain more than one module; the header of the main module *must* be as follows:

```
module fpmultiplier(output logic [31:0] product, output logic ready,
                    input logic [31:0] a, input logic clock, nreset);
```

The second file is a brief report of 2 pages, Design_Report.pdf. You can create this PDF file using either the Word or Latex templates provided. Please check that the PDF file opens correctly in Acrobat.

You may submit multiple versions of your SystemVerilog code and the final report. The code in each submission will be automatically marked and the score for these submissions will be emailed back to you, but only the score of the last version will be used in the final assessment.

If you submit more than 5 versions of your code, you will lose one mark for each further submission. In other words, if you submit 6 versions of your code (including the final version), you will lose 1 mark; if you submit 7 versions, you will lose 2 marks, and so on.

The deadline for submission is 16:00 on Tuesday 30 November 2021. Late submissions will be penalised in line with ECS regulations. Please ensure that your report file is included with your final submission.

Appendix: Behavioural Model

The following code is a functionally-equivalent, behavioural model of the multiplier. You can use this to verify your design, by comparing the bit pattens. This model completes in one clock cycle – your design is likely to require more cycles – but meets the timing specification.

Note. You *must not* use the `real` or `shortreal` types in your design. All submitted designs will be checked for these types before simulation and if the test fails, will not be simulated or synthesised. In other words you will receive *zero* marks.

```
module fpmultiplier_behav (output logic [31:0] product, output logic ready,
                      input logic [31:0] a, input logic clock, nreset);

// to allow inputs to be loaded in succession
enum {start, loada, loadb} state;
shortreal rproduct, ra; // shortreal is 32 bit floating point

always @(*)
  product = $shortrealtobits(rproduct); // converts real to bits

always @(*)
  ra = $bitstoshortreal(a); // converts bits to real

always @(posedge clock, negedge nreset)
if (~nreset)
  begin
  rproduct <= 0;
  state <= start;
  end
else
  begin
// This is a state machine, but there are other ways to do this
  case (state)
    start : begin
            state <= loada;
            end
    loada : begin
            rproduct <= ra;
            state <= loadb;
            end
    loadb : begin
            rproduct <= rproduct * ra;
            state <= start;
            end

  endcase
  end

 always @(*)
    ready = (state == start);

endmodule
```

An outline testbench can be used to simulate this design – and yours.

```
module test_fpmultiplier;

logic [31:0] product;
logic ready;
logic [31:0] a;
logic clock, nreset;
shortreal reala, realproduct;

fpmultiplier_behav a1 (.*);

initial
  begin
  nreset = '1;
  clock = '0;
  #5ns nreset = '1;
  #5ns nreset = '0;
  #5ns nreset = '1;
  forever #5ns clock = ~clock;
  end

initial
  begin
  @(posedge clock); // wait for clock to start
  @(posedge ready); // wait for ready
  @(posedge clock); //wait for next clock tick
  reala = 1.0;
  a = $shortrealtobits(reala);
  @(posedge clock);
  reala = -2.0;
  a = $shortrealtobits(reala);
  @(posedge ready);
  realproduct = $bitstoshortreal(product);
  $display("%f\n", realproduct);
  end
endmodule
```