ECE 5462: Verification Report 1

Matt Daehn, Cameron Allen, Haden Santefort, Stephen Parker, Christian Eakins, Youngsoo Kang

ECE 5462: HDL Design and Verification
Dr. Joanne Degroat
November 20, 2019

Contents

Abstract

In this report, the results of the verification plan for a floating point adder will be presented. The adder has four inputs: two IEEE Standard 754 single precision numbers, a latch signal to indicate when to read the inputs, and a drive signal to indicate when a result should be driven to the output bus. During testing the adder succeeded in outputting the correct value to the bus for all edge cases; however, exhibited errors whenever a fractional value was used as an input. The remainder of the document will flesh out the inputs and test bench used to debug the adder.

I. Overview

THE unit to be verified is a floating point adder with four inputs and one output. The inputs consist of two 32-bit std_logic_vectors for the two operands and two one bit std_logic values, latch and drive, to determine when the unit will latch the two thirty-two bit inputs and when the unit will drive the output of its operation onto the bus. The output is a 32-bit std_logic_vector that drives the output of the unit to the bus. The testbench will provide input vectors covering normal and edge cases and the expected result of the operation of the unit on said combinations.

II. Parts Verified

A. *Input Latches / Input Adjust*

The test bench will check that when the latch signal is asserted the input vectors are sent through to the unit and correct flags are set for each of the inputs. After checking that the correct values have been latched the testbench will check that the unit has successfully adjusted the inputs by putting them into normalized format. This section will also check if the input falls into any of the special cases such as NaN input, two opposite signed infinite inputs, one infinite input, or one zero input that the output signal is set to the correct value and the unit doesn't go into normal adding.

B. *Output Drive*

The testbench will check that when the drive signal is asserted the calculated output is driven to the output line, and at all other times when the drive signal is not asserted the output line will hold high impedance. The drive signal will be separated from the latch signal by enough time such that the unit can finish all calculations, but not much further than the expected time to ensure that the device can operate relatively quickly. This same time separation will be used for all inputs to check if the unit operates consistently even for edge cases. The output line must read the same correct output for the entire duration of the drive signal.

III. Generating the Testbench

A. *Input Vectors*

The input vectors provided by the testbench will encompass all possible input types and edge cases involving each type. For two normalized inputs, one normalized input and one denormalized input, and two denormalized inputs the testbench will cover cases involving combinations of positive, negative, whole, and decimal numbers including special cases that result in denormalized, $\pm\infty$, ± 0, and NaN outputs. The test bench will also cover special cases involving inputs of $\pm\infty$, ± 0, and NaN numbers in combination with each other and normalized/denormalized numbers.

B. *Input Vector Generation*

The input vectors were randomly generated using $Python$ 3 for robust testing (Appendix B). The randomizing was constrained to produce certain types of floating point, such as normalized or denormalized. The program uses two input lists that correspond to the inputs given to the adder and generates the expected output. It then creates a file that can be used in a VHDL test suite for testing purposes. However, manually generated and appended values were used for specific boundary conditions such as addition to $\pm\infty$ or ± 0.

C. *Testbench Code*

The code for the test bench is separated into two main parts, reading in the input vectors and checking the output. For reading in the vectors, the code reads in the first six characters for the input id and then the next 32 characters for the input value. This format is utilized twice on the first line to read in the two inputs and once on the next line for the expected output (III-C1, below). This pattern is repeated until all lines in the input file have been read into the program.

1) Read Input Vectors:

```
WHILE (NOT ENDFILE(test_data)) LOOP
   --get next input test vector and expected result
   readline(test_data,cur_line);
   read(cur_line,aid); read(cur_line,a_test_val);
   read(cur_line,bid); read(cur_line,b_test_val);
   readline(test_data,cur_line);
   read(cur_line,resid);read(cur_line,result_val);
   std_result_val := To_StdLogicVector(result_val);
   num_tests := num_tests + 1;
```

The second part of the code sends the input values out to the adder while also setting the drive and assert signals at specific time intervals. The code then waits for a set amount of time for the output from the adder to be driven to C at which point it compares C to the expected output from the vector file and sets the error signal accordingly (III-C2, below). When we originally discussed the scope of the test bench in the plan, we described adding checks for specific components such as the renormalization section; however, after looking over the dataflow code we were debugging we realized that such a scope would require us to modify the dataflow's code so we decided to limit the scope to just checking the input and output.

2) Send Input Vectors to Adder:

```
-- run through bus cycle to send data to unit
   aid_sig <= "======", aid after 20 ns;
   bid_sig <= "======", bid after 20 ns;
   resid_sig <= "======", resid after 20 ns;
   -- drive signals on bus
   aval <= To_StdLogicVector(a_test_val) after 20 ns, HIGHZ after 80 ns;
   bval <= To_StdLogicVector(b_test_val) after 20 ns, HIGHZ after 80 ns;
   latch <= '0' after 20 ns, '1' after 70 ns;
   wait for 100 ns;
   drive <= '0' after 20 ns, '1' after 80 ns;
   exp_res <= std_result_val after 20 ns, HIGHZ after 80 ns;
   wait for 50 ns;
   ASSERT (C = std_result_val)
     REPORT "result does not agree with expected result"
     SEVERITY WARNING;
   IF (C /= std_result_val) THEN
     num_errors := num_errors + 1;
     err_sig <= '1', '0' after 10 ns;
     err_count <= num_errors;
   END IF;
   wait for 50 ns;
  END LOOP;
```

IV. Examining Results

A. Non-Issues

Almost all of the test cases seemed to yield the correct answer. When special cases were given (NaN, $\pm\infty$, and denormal numbers), the results were what we expected, which were NaN for any input that contained NaN or if one was $+\infty$ and one was $-\infty$, $\pm\infty$ when the inputs were $\pm\infty$, or the other input when given a denormalized number. When given normalized integer values, the results were what was expected; the integer addition between the two inputs. For these values, we have deemed that the adder was successful.

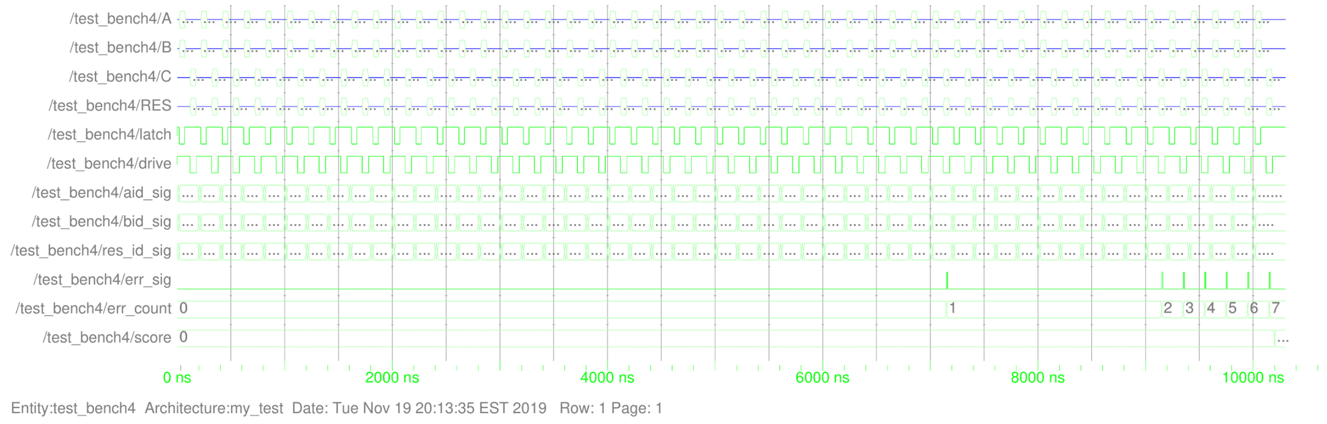

Fig. 1. Overview of testbench waveform output.

B. Issues

Of the 51 test cases that were input to the unit under test, seven failed. However, all of these failed test cases have to do with the rounding of the least significant binary place. For example, one of the test cases that failed was the addition of 100 and 0.01. The expected result, in hexadecimal and single point floating precision, 0x42C8051E. The actual result generated by the adder was 0x42C8051F. When translated to decimal numbers, this is equivalent to the difference between 100.00999 and 100.01000. For most practical purposes, this distinction is not especially important, and no other significant errors were found in the adder.

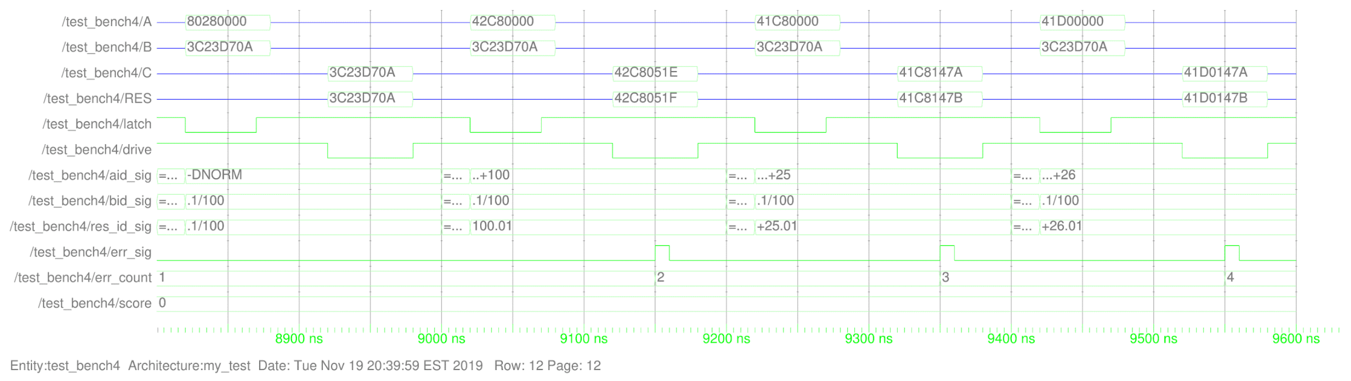

Fig. 2. Failed cases of testbench waveform output.

Appendix A
VHDL Code

fpatb.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use WORK.fpa_support.all;
use STD.TEXTIO.all;
use WORK.fpa_test_vect.ALL;
entity test_bench4 is
end test_bench4;

architecture my_test of test_bench4 is
  signal A,B,C,RES : std_logic_vector (31 downto 0);
  signal latch,drive : std_ulogic;
  signal aid_sig,bid_sig,res_id_sig : string (1 to 6);
  signal err_sig : bit;
  signal score : integer := 0;
  signal err_count : integer := 0;

COMPONENT fpa
  PORT (A,B : IN std_logic_vector (31 downto 0);
       latch, drive: IN std_ulogic;
       C : OUT std_logic_vector (31 downto 0));
END COMPONENT;
FOR ALL : fpa USE ENTITY WORK.fpa(data_flow);

BEGIN

a1 : fpa PORT MAP(A,B,latch,drive,C);

gen_vec(aid_sig,bid_sig,res_id_sig,A,B,RES,C,latch,drive,err_sig,score,err_count);

END my_test;
```

test_vect.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use STD.TEXTIO.all;
package fpa_test_vect is

constant HIGHZ : std_logic_vector (31 downto 0)
        := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

procedure gen_vec (SIGNAL aid_sig,bid_sig,resid_sig : OUT string (1 to 6);
             SIGNAL aval,bval,exp_res : OUT std_logic_vector (31 downto 0);
             SIGNAL C : IN std_logic_vector (31 downto 0);
             SIGNAL latch,drive : OUT std_ulogic;
             SIGNAL err_sig : OUT bit;
             SIGNAL score : OUT integer;
             SIGNAL err_count : OUT integer);

end fpa_test_vect;

package body fpa_test_vect is

procedure gen_vec (SIGNAL aid_sig,bid_sig,resid_sig : OUT string (1 to 6);
             SIGNAL aval,bval,exp_res : OUT std_logic_vector (31 downto 0);
             SIGNAL C : IN std_logic_vector (31 downto 0);
             SIGNAL latch,drive : OUT std_ulogic;
             SIGNAL err_sig : OUT bit;
             SIGNAL score : OUT integer;
             SIGNAL err_count : OUT integer) is
variable cur_line : LINE;
file test_data: TEXT is IN "/home/kang.676/HDLProj/vp1/testvectors";
variable a_test_val, b_test_val : bit_vector (31 downto 0);
variable result_val : bit_vector (31 downto 0);
variable std_result_val : std_logic_vector (31 downto 0);
variable aid,bid,resid : string (1 to 6);
variable num_tests,num_errors : integer := 0;
Begin
  aval <= HIGHZ; bval <= HIGHZ; exp_res <= HIGHZ;
  latch <= '1'; drive <= '1';
  WHILE (NOT ENDFILE(test_data)) LOOP
    --get next input test vector and expected result
    readline(test_data,cur_line);
    read(cur_line,aid); read(cur_line,a_test_val);
    read(cur_line,bid); read(cur_line,b_test_val);
    readline(test_data,cur_line);
```

```
   read(cur_line,resid);read(cur_line,result_val);
   std_result_val := To_StdLogicVector(result_val);
   num_tests := num_tests + 1;
   -- run through bus cycle to send data to unit
   aid_sig <= "======", aid after 20 ns;
   bid_sig <= "======", bid after 20 ns;
   resid_sig <= "======", resid after 20 ns;
   -- drive signals on bus
   aval <= To_StdLogicVector(a_test_val) after 20 ns, HIGHZ after 80 ns;
   bval <= To_StdLogicVector(b_test_val) after 20 ns, HIGHZ after 80 ns;
   latch <= '0' after 20 ns, '1' after 70 ns;
   wait for 100 ns;
   drive <= '0' after 20 ns, '1' after 80 ns;
   exp_res <= std_result_val after 20 ns, HIGHZ after 80 ns;
   wait for 50 ns;
   ASSERT (C = std_result_val)
     REPORT "result does not agree with expected result"
     SEVERITY WARNING;
   IF (C /= std_result_val) THEN
     num_errors := num_errors + 1;
     err_sig <= '1', '0' after 10 ns;
     err_count <= num_errors;
   END IF;
   wait for 50 ns;
 END LOOP;
 score <= ((num_tests - num_errors)* 100)/num_tests;
 wait for 100 ns;
 wait;
END gen_vec;

end fpa_test_vect;
```

APPENDIX B
TEST VECTORS AND GENERATION CODE

Test Vectors

```
...NaN 01111111100000000000000000000001...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001.+INIF 01111111100000000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001.-INIF 11111111100000000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001....+0 00000000000000000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001....-0 10000000000000000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001...+25 01000001110010000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001...-25 11000001110010000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001..+100 01000010110010000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001..-100 11000010110010000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001.1/100 00111100001000111101011100001010
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001+DNORM 00000000010100000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001-DNORM 10000000010100000000000000000000
...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
.+INIF 01111111100000000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
.-INIF 11111111100000000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
```

```
....+0 00000000000000000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
....-0 10000000000000000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
...+25 01000001110010000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
...-25 11000001110010000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
..+100 01000010110010000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
..-100 11000010110010000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
.1/100 00111100001000111101011100001010...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
+DNORM 00000000010100000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
-DNORM 10000000010100000000000000000000...NaN 01111111100000000000000000000001
...NaN 01111111100000000000000000000001
.+INIF 01111111100000000000000000000000.+INIF 01111111100000000000000000000000
.+INIF 01111111100000000000000000000000
.+INIF 01111111100000000000000000000000.-INIF 11111111100000000000000000000000
...NaN 01111111100000000000000000000001
.-INIF 11111111100000000000000000000000.-INIF 11111111100000000000000000000000
.-INIF 11111111100000000000000000000000
...+25 01000001110010000000000000000000....+0 00000000000000000000000000000000
...+25 01000001110010000000000000000000
...+25 01000001110010000000000000000000....-0 10000000000000000000000000000000
...+25 01000001110010000000000000000000
....+0 00000000000000000000000000000000..+100 01000010110010000000000000000000
..+100 01000010110010000000000000000000
....-0 10000000000000000000000000000000..-100 11000010110010000000000000000000
..-100 11000010110010000000000000000000
...+25 01000001110010000000000000000000..+100 01000010110010000000000000000000
..+125 01000010111110100000000000000000
..+100 01000010110010000000000000000000...+25 01000001110010000000000000000000
..+125 01000010111110100000000000000000
..+100 01000010110010000000000000000000..-100 11000010110010000000000000000000
....+0 00000000000000000000000000000000
..-100 11000010110010000000000000000000..+100 01000010110010000000000000000000
....+0 00000000000000000000000000000000
..+100 01000010110010000000000000000000.1/100 00111100001000111101011100001010
100.01 01000010110010000000010100011111
+DNORM 00000000010100000000000000000000..+100 01000010110010000000000000000000
..+100 01000010110010000000000000000000
-DNORM 10000000010100000000000000000000..-100 11000010110010000000000000000000
..-100 11000010110010000000000000000000
....+1 00111111100000000000000000000000...+25 01000001110010000000000000000000
...+26 01000001110100000000000000000000
....-1 10111111100000000000000000000000...+25 01000001110010000000000000000000
...+24 01000001110000000000000000000000
...NaN 01111111100000000000000000000001.1/100 00111100001000111101011100001010
...NaN 01111111100000000000000000000001
.+INIF 01111111100000000000000000000000.1/100 00111100001000111101011100001010
.+INIF 01111111100000000000000000000000
.-INIF 11111111100000000000000000000000.1/100 00111100001000111101011100001010
.-INIF 11111111100000000000000000000000
+DNORM 00000000010100000000000000000000.1/100 00111100001000111101011100001010
.1/100 00111100001000111101011100001010
-DNORM 10000000010100000000000000000000.1/100 00111100001000111101011100001010
.1/100 00111100001000111101011100001010
..+100 01000010110010000000000000000000.1/100 00111100001000111101011100001010
100.01 01000010110010000000010100011111
...+25 01000001110010000000000000000000.1/100 00111100001000111101011100001010
+25.01 01000001110010000001010001111011
...+26 01000001110100000000000000000000.1/100 00111100001000111101011100001010
+26.01 01000001110100000001010001111011
...+24 01000001110000000000000000000000.1/100 00111100001000111101011100001010
+24.01 01000001110000000001010001111011
...+24 01000001110000000000000000000000.+1/10 00111101110011001100110011001101
```

```
.+24.1 01000001110000000110011001100110 1
...+25 01000001110010000000000000000000.+1/10 00111101110011001100110011001101
.+25.1 01000001110010001100110011001101
```

Python 3 Code for Test Vector Generation[1]

1) AutoGen1.py:

```
"""
   Test case auto generator for ECE 5462 Verification project 1.
   ********************************************
   * This file does not need to be updated or *
   * changed. Only change lists.py, never *
   * this one.                          *
   ********************************************
   Written by: Haden Santefort
   Generates test cases for use in the VHDL verification.
   Written and tested using Python version 3.6.8
   Change Log:
      (11/11/19)
      1. Several additions have been created. It should be OK depending on how we
      want to define success. Will add more if we decide to
      2. Haven't dealt with denormalized numbers because I'm still not 100% sure
      how it's supposed to handle those. Clearly will discuss that and determine
      the proper method moving forward
      3. The write function is completed and doesn't need any further alteration.
      getResults and parseName will need to be constantly updated as we add more
      test cases and finalize everything. Currently we have 36 test cases, most
      of which are NaN and +/- infinity results. A couple of basic addition test
      have been added as well. I think all that we have left is to figure
      out denormal numbers and their respective results
      4. Fixed formatting to match fpmvectors from the multiplier project step
      (11/11/19 @ 1:20 PM)
      1. Added denorm result. I believe if there is a denormalized number you just get the
      other input. Will confirm later
      (11/14/19)
      1. Added positive and negative one, as well as pos 24 and 26 for testing
      2. Moved test variables to list.py to improve readability and reusability between
      the python3 and python2 versions
      3. Moved functions that parse the data to lists to improve reusability between python3
         and
      python2 versions
      (11/18/19)
      1. Refactored parseName to use a dictionary instead of defining each value as a seperate
      variable to improve readability and useability. Makes it more DRY.
"""

from typing import List
from lists import *
import sys
array = List[str]

# Input values
inputA = [NaN, pos_infin, neg_infin, pos_dnorm, neg_dnorm, pos_100, pos_25, pos_26, pos_24,
    pos_24, pos_25]
inputB =
    [pos_hundredth,pos_hundredth,pos_hundredth,pos_hundredth,pos_hundredth,pos_hundredth,pos_hundredth,po
    pos_tenth, pos_tenth]

def generateOutput(inputA: array, inputB: array) -> array:
   """
      Generates the output list based on the input lists.
      inputA: Inputs at input A
      inputB: Inputs at B
   """
   output = []
```

[1] $https://github.com/hsantefort12/Verification\_Generator$

```
   for data in zip(inputA, inputB):
      if (data[0] == NaN or data[1] == NaN):
         output.append(NaN)
      elif (data[0] == pos_infin or data[1] == pos_infin):
         output.append(pos_infin)
      elif (data[0] == neg_infin or data[1] == neg_infin):
         output.append(neg_infin)
      elif ((data[0] == neg_infin and data[1] == pos_infin) or (data[1] == neg_infin and
          data[0] == pos_infin)):
         output.append(NaN)
      else:
         o = getResult(data[0], data[1])
         output.append(o)
   return output

filename = "noname" if len(sys.argv) < 2 else sys.argv[1]
output = generateOutput(inputA, inputB)
write(inputA, inputB, output, filename, "add")
o = "Successfully created file in file name {0} with {1} test cases."
print(o.format(filename, len(output)))
```

2) *List.py:*

```
"""
   This file contains all necessary constants, lists, and parsing
   functions needed for auto generating test vectors.
   This is reused between autoGen1 for verification assignment 1
   and autoGen2 for verification assignment 2
   Written by: Haden Santefort
   Conversions received using: http://weitz.de/ieee/
"""

from typing import List
array = List[str]

# Constant numeric values. Used for dictionary keys
NaN = "NaN"
pos_infin = "+INIF"
neg_infin = "-INIF"
pos_zero = "+0"
neg_zero = "-0"
pos_25 = "+25"
neg_25 = "-25"
pos_100 = "+100"
neg_100 = "-100"
pos_hundredth = "1/100"
pos_dnorm = "+DNORM"
neg_dnorm = "-DNORM"
pos_125 = "+125"
pos_100_01 = "100.01"
pos_one = "+1"
neg_one = "-1"
pos_26 = "+26"
pos_24 = "+24"
pos_25_01 = "+25.01"
pos_26_01 = "+26.01"
pos_24_01 = "+24.01"
pos_tenth = "+1/10"
pos_25_1 = "+25.1"
pos_24_1 = "+24.1"

# Dictionary containing binary values for addition
valuesAdd = {
   NaN: "01111111100000000000000000000001",
   pos_infin: "01111111100000000000000000000000",
   neg_infin: "11111111100000000000000000000000",
   pos_zero: "00000000000000000000000000000000",
   neg_zero: "10000000000000000000000000000000",
   pos_25: "01000001110010000000000000000000",
   neg_25: "11000001110010000000000000000000",
   pos_100: "01000010110010000000000000000000",
   neg_100: "11000010110010000000000000000000",
   pos_hundredth: "00111100001000111101011100001010",
   pos_dnorm: "00000000010100000000000000000000",
   neg_dnorm: "10000000010100000000000000000000",
   pos_125: "01000010111110100000000000000000",
   pos_100_01: "01000010110010000000010100011111",
   pos_one: "00111111100000000000000000000000",
   neg_one: "10111111100000000000000000000000",
   pos_26: "01000001110100000000000000000000",
   pos_24: "01000001110000000000000000000000",
   pos_25_01: "01000001110010000001010001111011",
   pos_26_01: "01000001110100000001010001111011",
   pos_24_01: "01000001110000000001010001111011",
   pos_tenth: "00111101110011001100110011001101",
   pos_25_1: "01000001110010001100110011001101",
   pos_24_1: "01000001110000001100110011001101"
}

# Special cases for adder
specialCases = [NaN, pos_infin, neg_infin, pos_dnorm, neg_dnorm]

# Holds multiplication results
valuesMult = {}

# Expected output length (int) for multiplier
LENGTH = 46

#--------------------- End Constants-------------------------

def formatNameMult(name: str) -> str:
   """
      Formats names to be 6 charactes for the
      multiplier input.
   """
   if len(name) <= 6:
      dots = ""
      for i in range(0, 6 - len(name)):
         dots += '.'
      return dots + name
   else:
      return name[0] + 'BIG'


def formatNameAdd(name: str) -> str:
   """
      Formats the adder inputs to be
      6 characters long. Do not have
      to worry about larger values
      here.
   """
   dots = ""
   for i in range(0, 6 - len(name)):
      dots += '.'
   return dots + name

def write(inputA: array, inputB: array, output: array, filename: str, multOrAdd: str):
   """
      Writes to the test vector file in filename.
      inputA: Array of the input names.
      inputB: Array of input names.
      output: Array of output values
      filename: Name of the file
   """
   values = valuesMult if multOrAdd == "mult" else valuesAdd
   f = open(filename, 'w')
   for values in zip(inputA, inputB, output):
      for i, d in enumerate(values):
         result = parseName(d, multOrAdd, i)
         if result[:11] == "Fatal Error":
            print(result + ". Invalid input")
            exit()
         if multOrAdd == "mult":
            f.write(formatNameMult(str(d)) + " " + result)
         else:
            f.write(formatNameAdd(str(d)) + " " + result)
         if (i == 1):
            f.write('\n')
      f.write('\n')
   f.close()

def trimName(name: str, i: int, addOrMult: str) -> str:
   """
      Trims input names for the multipler.
      Sometimes the same value for input and output
      exists and need to be a different length
   """
   if i == 2 or addOrMult == "add" or len(name) <= (LENGTH // 2):
      return name
   return name[(LENGTH // 2):]

def parseName(name: str, addOrMult: str, i: int):
   """
      Parses the names. Essentially gets the name from
      the dictionary and trims them as necessary.
   """
   values = valuesMult if addOrMult == "mult" else valuesAdd
   if name in values:
      return trimName(values[name], i, addOrMult)
   else:
      return "Fatal error: {} doesn't exist".format(name)


def getResult(a, b):
   """
      Gets the result of anything that isn't a NaN, pos infinity, or negative
      infinity for the floating point adder test cases.
      a: string of input a
      b: string of input b
      return: returns a string of the value of the result
   """
   if (a == pos_zero or a == neg_zero):
      return b
   if (b == pos_zero or b == neg_zero):
      return a
   if ((a == pos_100 or b == pos_100) and (a == pos_25 or b == pos_25)):
      return pos_125
   if ((a == pos_100 or b == pos_100) and (a == neg_100 or b == neg_100)):
      return pos_zero
   if ((a == pos_100 or b == pos_100) and (a == pos_hundredth or b == pos_hundredth)):
      return pos_100_01
   # TODO: Enusre that this is the correct implementation
   if (a == pos_dnorm or a == neg_dnorm):
      return b
   if (b == pos_dnorm or b == neg_dnorm):
      return a
   if ((a == pos_one and b == pos_25) or (a == pos_25 and b == pos_one)):
      return pos_26
   if ((a == neg_one and b == pos_25) or (a == pos_25 and b == neg_one)):
      return pos_24
   if ((a == pos_24 or b == pos_24) and (a == pos_hundredth or b == pos_hundredth)):
      return pos_24_01
   if ((a == pos_25 or b == pos_25) and (a == pos_hundredth or b == pos_hundredth)):
      return pos_25_01
   if ((a == pos_26 or b == pos_26) and (a == pos_hundredth or b == pos_hundredth)):
      return pos_26_01
   if ((a == pos_25 or b == pos_25) and (a == pos_tenth or b == pos_tenth)):
      return pos_25_1
   if ((a == pos_24 or b == pos_24) and (a == pos_tenth or b == pos_tenth)):
      return pos_24_1
```