Three-stage Pipelined Half-Precision (HP) Floating-Point (FP) Adder

EE271 Project Report

Guided by,

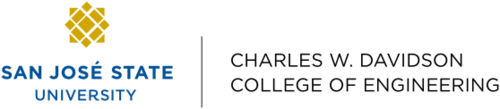
Prof: Binh Le

*Presented* By:

Gaurav Deore (ID: 015215746)

Isha Baokar (ID: 015336841)

Mokshith Raja (ID: 015351765)



**One Washington Square, San José, CA 95192-0080.**

**INDEX**

**Contents**

* Abstract ……………………………………………………………………………………………. 1
* Introduction ………………………………………………………………………………………

1. IEEE Floating Point Algorithm ………………………………………………………….
2. IEEE Floating Point Architecture ……………………………………………………...
3. 3-Stage Pipelined Floating Point Adder …………………………………………..

* Pin Level Implementation …………………………………………………………………
* Hardware Components Used …………………………………………………………...
* Software Components Used …………………………………………………………….
* State Diagram ………………………………………………………………………………….
* Working …………………………………………………………………………………………..
* Simulation Results …………………………………………………………………………..
* Conclusion ………………………………………………………………………………………
* Credits …………………………………………………………………………………………….
* Annexure ………………………………………………………………………………………..

Abstract

The designed Floating-Point Adder is a 16-bit or half precision floating point adder. The working of the adder is divided into states which are implemented using Finite State Machine (FSM) logic. It takes two inputs in hexadecimal format through a numerical keypad and does the required calculation using the floating-point addition logic. The following project is implemented using FSM with four states, being Reset, Input\_1, Input\_2 and FP\_Calculation. The setup includes and LCD display for displaying the current\_state of the FSM controlled by the Arduino. The floating- point code is written in Verilog and tested in ModelSim. It is simulated and implemented on Terasic DE-10 lite FPGA using Quartus Prime Lite Edition.

Introduction

Floating point addition involves encoding inputs and outputs to properly represent the different parts of a number represented in scientific notation. Further, the addition requires a nontrivial algorithm to properly decode, add, and re-encode the floating-point values.

1. Floating Point Encoding

The inputs and outputs are encoded according to the “binary16” interchange format, specified in IEEE Standard 754-20081, and illustrated in Table 1.



This encoding represents a number whose value is given in the below Equation –



* Sign: The sign bit consists of a single bit. It can take one of the two values '0' or '1'. '0' indicates a positive number and '1' indicates a negative number
* Exponent: The exponent consists of five bits. It indicates the exponent value with the bias.
* Significand: The significand consists of eleven bits which includes a hidden bit. If significand is represented as 'x', the representation for a normalized number is '1.x'.
* Range: Minimum value of Exponent is -14 and the Maximum value of Exponent is 15. The bias for this format= 15 (Any value outside the range will be indicated by raising the exception flags)
* True exponent value = Represented exponent - Bias
* Example: If we represent exponent as 01101 = 1+4+8 =13, the actual/true exponent value is 13 - 15 which is -2.

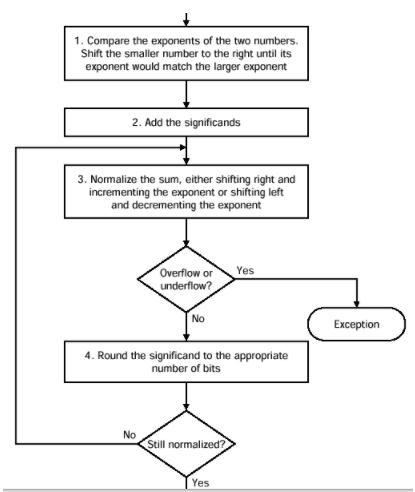


Figure Floating Point

1. Floating -Point Conversion

To convert the decimal number into the single-precision and double-precision floating-point numbers following procedure is carried out with the example:

* Choose the floating-point data format (single-precision or double-precision)
* Separate the whole and decimal part of number
* Convert the whole and decimal into binary
* Combine the two parts of the number that have been converted into binary
* Convert the binary number into base 2 scientific notation 6. Determine the sign of the number and display in binary format
* Get the exponent based on precision
* Convert the exponent into binary
* Determine the mantissa

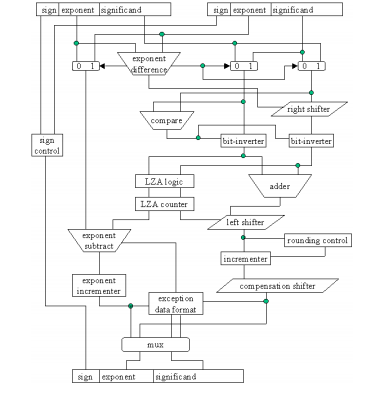
1. IEEE Floating Point Algorithm

The addition of the two half-precision floating-point numbers is carried out using following steps:

* Compare the exponents of two numbers for (or) and calculate the absolute value of difference between the two exponents (). Take the larger exponent as the tentative exponent of the result.
* Shift the significand of the number with the smaller exponent, right through a number of bit positions that is equal to the exponent difference. Two of the shifted-out bits of the aligned significand are retained as guard (G) and Round (R) bits. So, for p bit significands, the effective width of aligned significand must be p + 2 bits. Append a third bit, namely the sticky bit (S), at the right end of the aligned significand. The sticky bit is the logical OR of all shifted out bits. Step 3) Add/subtract the two signed-magnitude significands using a p + 3-bit adder. Let the result of this is SUM.
* Check SUM for carry out (Cout) from the MSB position during addition. Shift SUM right by one bit position if a carry out is detected and increment the tentative exponent by 1. During subtraction, check SUM for leading zeros. Shift SUM left until the MSB of the shifted result is a 1. Subtract the leading zero count from tentative exponent. Evaluate exception conditions, if any.
* Round the result if the logical condition R” (M0 + S’’) is true, where M0 and R’’ represent the pth and (p + 1) st bits from the left end of the normalized significand. New sticky bit (S’’) is the logical OR of all bits towards the right of the R’’ bit. If the rounding condition is true, a 1 is added at the pth bit (from the left side) of the normalized significand. If p MSBs of the normalized significand are 1’s, rounding can generate a carry-out. in that case normalization (step 4) has to be done again.

1. IEEE Floating Point Algorithm

The 16-bit half-precision floating-point addition of two numbers is performed using the following architecture



1. 3-Stage Pipelined Floating Point Adder

The 3-stage floating-point addition is performed as described below:

* Step-1 - Compare the exponents and determine the number of shifts required to align the mantissa to make the exponents equal. Then, right-shift the mantissa of the smaller exponent by the required amount to normalize the mantissa. After combinational computation, the result is stored into pipeline buffer register.
* Step-2 - Compare the two aligned mantissas and determine which is the smaller of the two. Take 2’s complement of the smaller mantissa if the signs of the two numbers are different. Then, add the two mantissas. (addition).
* Step-3 - Determine the number of shifts required and the corresponding direction to normalize the result and shift the mantissa to the required direction by the required amount. Adjust the exponent accordingly and check for any exceptional condition.

Pin-Level Implementation

The floating-point adder implementation is carried out using several hardware devices and peripherals connected in an orderly manner to get the desired output. For the implementation of the design, we have used a 4X4 numerical keypad, LCD interfaced with an Arduino board and DE-10Lite FPGA Board.

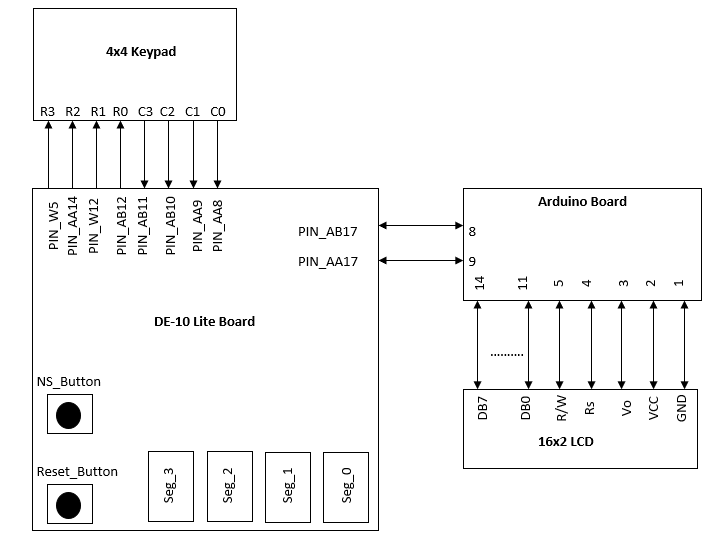


Figure Pin Level Representation

The 4X4 numerical keypad is interfaced with the DE-10 lite via GPIO pins ranging from GPIO [10][12][14][16][18][20][22][24] with each going to the subsequent row and column configuration of the keypad. The Arduino Board is connected to the FPGA via the Arduino expansion header PIN\_AB17 and PIN\_AA17 which connect to the Arduino\_IO8 and Arduino\_IO9 respectively. The LCD is interfaced with the Arduino board, with pins DB\_0 to 7 acting as the input to the data lines, this input will be treated as data to be displayed on the LCD screen, the R/W pin is connected to pin 5 of the Arduino, this is used to select between read and write modes, the RS pin is connected to the digital pin 12 of the Arduino. The LCD and the Arduino are interfaced in a 4-bit mode. The LCD and the Arduino were connected using jumper wires.

Hardware Components Used

The floating-point adder implementation used several hardware components, communicating with each other for the smooth execution of the pipelined floating-point adder. The various hardware tools used were as described below.

* DE-10 Lite FPGA Board- The logic of the floating-point adder was implemented on the FPGA using it’s several options for different functions such as a push button for changing the state of the FSM i.e., for going from reset to input1 to input2 and finally calculating the floating-point addition of the two numbers and displaying the value in HEX using the 7-segment display on the FPGA board. Apart from these the GPIO headers were used for connecting the 4X4 numerical keypad to the FPGA, the Arduino UNO R3 expansion header was used to connect the Arduino to the FPGA board.
* 4X4 Numerical Keypad – A 4X4 numerical keypad was used to enter the values from the user in the form of input1 and input2 on which the floating-point addition algorithm would be performed. The used 4X4 numerical keypad is a membrane-based keypad, it has 16 keys and beneath each key is a special membrane switch, it has 4 rows and 4 columns. Pressing a button shorts one of the row line to one of the column line allowing current to flow between them. This keypad was directly connected to the FPGA board via the GPIO connecter and used for inputting the values.
* Arduino Board – The Arduino board is a microcontroller board which has 14 digital input/output pins along with an USB connection, power jack and a reset button. The Arduino is interfaced with the FPGA board using the pins of the Arduino UNO R3 expansion header in the de-10 lite. The main purpose of the Arduino is to connect the LCD display. The LCD and Arduino are connected using jumper wires.
* LCD Display- The Liquid Crystal Display is used in the project to display the current state of the floating-point adder, such that the user knows what state is he in right now. The LCD is interfaced with the Arduino board in a similar manner.
* Jumper Wire- These are used to make connections between the Arduino board and the LCD display.

Software Used

The design implementation and logic verification for the floating- point adder is done in Quartus Prime. Quartus Prime is programmable logic device design software which helps in compilation of design and simulation of the design against a given stimuli and ultimately configure the target device with the programmer. The following design code is written in Verilog HDL and burnt into the DE-10 lite FPGA board.

State Diagram

The state diagram for the floating-point adder contains 4 states which are Reset, input-1, input-2 and FP Calculation

Working

The Floating-point adder is implemented using pipeline logic for a better throughput. The floating-point point adder basically does three important steps which are align, add and normalize. Below is a basic flowchart describing the stages involved in floating-point addition.

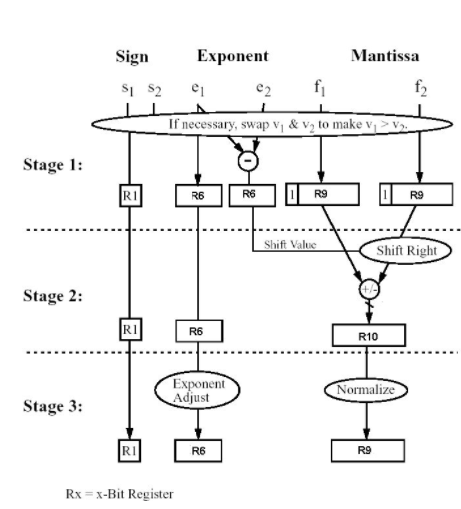
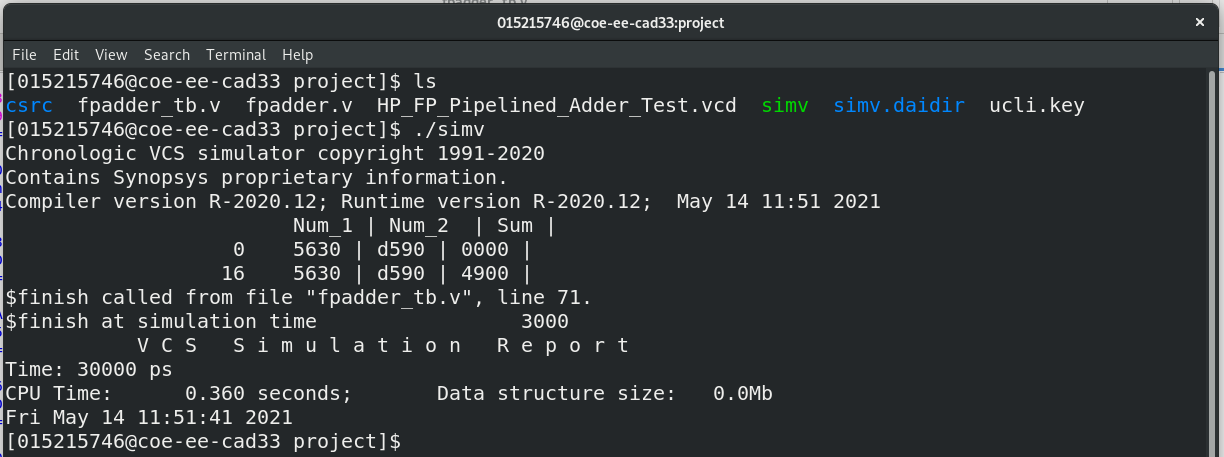


Figure Flow Chart for Floating Point Adder

# **Simulation Results:**

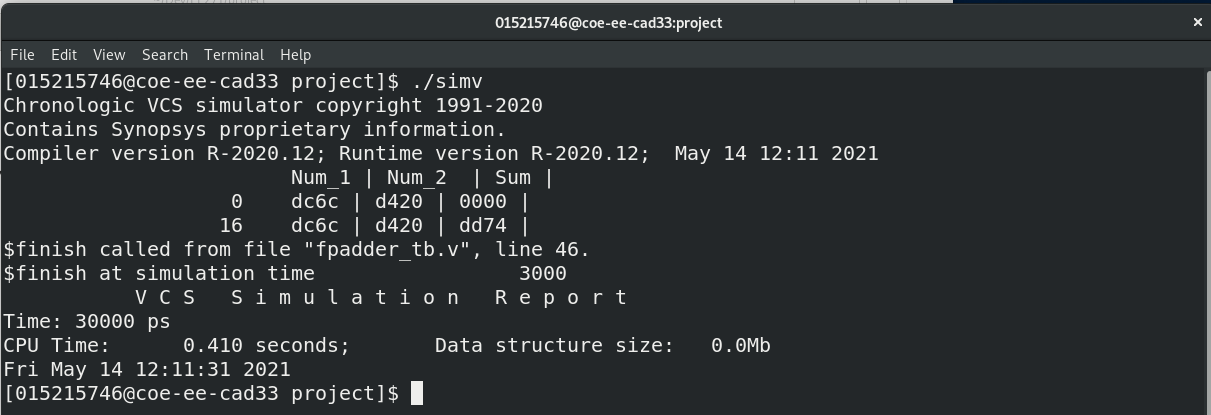
**Test case I)**

Input 1 = 16’h5630, Input 2 = 16’hD590 => Sum = 16’h4900



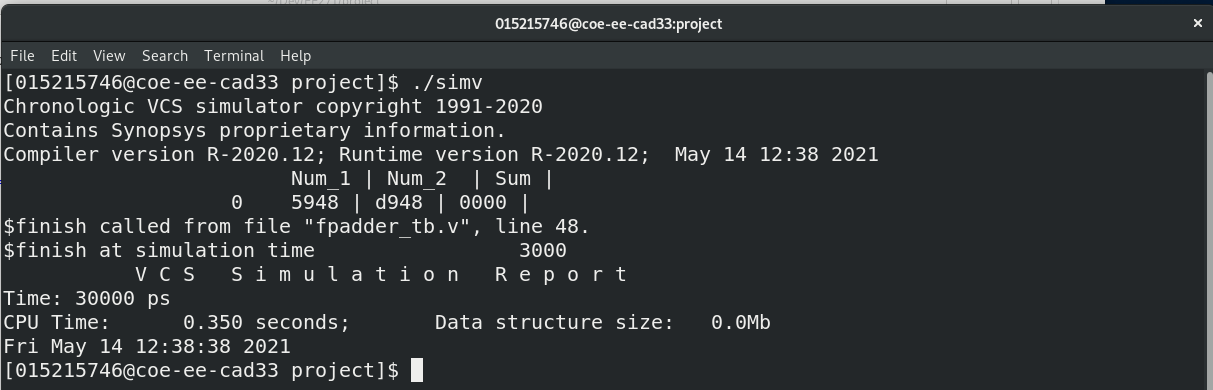
**Test case II)**

Input 1 = 16’hDC6C, Input 2 = 16’hD420 => Sum = 16’hDD74



**Test case III)**

Input 1 = 16’h5948, Input 2 = 16’hD948 => Sum = 16’h0000



# **Annexure**:

**Arduino Code:**

#include <LiquidCrystal.h>

#define pin8 8

#define pin9 9

int Contrast=40;

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup()

{

pinMode(8,INPUT);

pinMode(9,INPUT);

analogWrite(6,Contrast);

lcd.begin(16, 2);

}

void loop()

{

while ((digitalRead(pin8)==0) && (digitalRead(pin9)==0))

{

lcd.setCursor(0,0);

lcd.write("RESET STATE");

}

lcd.clear();

while((digitalRead(pin8)==0) && (digitalRead(pin9)==1))

{

lcd.setCursor(0,0);

lcd.write("INPUT 1 STATE");

}

lcd.clear();

while((digitalRead(pin8)==1) && (digitalRead(pin9)==0))

{

lcd.setCursor(0,0);

lcd.write("INPUT 2 STATE");

}

lcd.clear();

while((digitalRead(pin9)==1) && (digitalRead(pin8)==1))

{

lcd.setCursor(0,0);

lcd.write("FP OUTPUT");

}

lcd.clear();

}

**3-stage Pipelined Half-precision Floating Point Adder:**

`timescale 1ns/10ps

module HP\_FP\_Pipelined\_Adder(input clk, input reset, input [15:0] num1, input [15:0] num2, output [15:0] sum);

reg [15:0] shift\_46;

reg [4:0] larger\_expo\_46,sum\_expo\_46;

reg [9:0] small\_expo\_mant\_46,large\_expo\_mant\_46, S\_mant\_46, L\_mant\_46, large\_mant\_46, sum\_mant\_46;

reg [10:0] add\_mant\_46, add1\_mant\_46;

reg [3:0] denorm\_shift\_46;

integer signed denorm\_expo\_46;

reg [15:0] sum\_46;

reg s1\_46,s2\_46,sum\_sign\_46;

reg [4:0] e1\_46,e2\_46;

reg [9:0] m1\_46,m2\_46;

assign sum = sum\_46;

reg [4:0] larger\_expo\_pipe1\_46,larger\_expo\_pipe2\_46,larger\_expo\_pipe3\_46;

reg [9:0] S\_expo\_mant\_46,L\_expo\_mant\_46, S\_expo\_mant\_pipe1\_46, L\_expo\_mant\_pipe1\_46;

reg [10:0] add\_mant\_pipe2\_46,add\_mant\_pipe3\_46;

reg [3:0] denorm\_shift\_pipe3\_46;

integer signed denorm\_expo\_pipe3\_46;

reg s1\_pipe1\_46,s1\_pipe2\_46,s1\_pipe3\_46,s2\_pipe1\_46,s2\_pipe2\_46,s2\_pipe3\_46;

reg [4:0] e1\_pipe1\_46,e1\_pipe2\_46,e1\_pipe3\_46,e2\_pipe1\_46,e2\_pipe2\_46,e2\_pipe3\_46;

reg [9:0] m1\_pipe1\_46,m1\_pipe2\_46,m1\_pipe3\_46,m2\_pipe1\_46,m2\_pipe2\_46,m2\_pipe3\_46;

always @(posedge clk)

begin

// Extract sign bit, exponent and mantissa from both numbers

s1\_46 = num1[15]; //1 bit for SP

s2\_46 = num2[15];

e1\_46 = num1[14:10]; //8 bit for SP

e2\_46 = num2[14:10];

m1\_46 = num1[9:0]; // 23 bit for SP

m2\_46 = num2[9:0];

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Stage I)

//Compare the exponents and determine the amount of shift required

if(e1\_46 > e2\_46) // Here Expo\_1 is larger

begin

shift\_46 = e1\_46 - e2\_46;

larger\_expo\_46 = e1\_46;

small\_expo\_mant\_46 = m2\_46;

large\_expo\_mant\_46 = m1\_46;

end

else // Here Expo\_2 is larger

begin

shift\_46 = e2\_46 - e1\_46;

larger\_expo\_46 = e2\_46;

small\_expo\_mant\_46 = m1\_46;

large\_expo\_mant\_46 = m2\_46;

end

if(e1\_46 == 0 | e2\_46 == 0) // If any one of the exponent is zero, no need to shift

shift\_46 = 0;

else

shift\_46 = shift\_46;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Right shift the smaller exponent mantissa & the nnormalise both the mantissas

if(e1\_46 != 0)

begin

S\_expo\_mant\_46 = {1'b1,small\_expo\_mant\_46[9:1]};

S\_expo\_mant\_46 = (S\_expo\_mant\_46 >> shift\_46);

end

else

S\_expo\_mant\_46 = small\_expo\_mant\_46;

if(e2\_46 != 0)

L\_expo\_mant\_46 = {1'b1,large\_expo\_mant\_46[9:1]};

else

L\_expo\_mant\_46 = large\_expo\_mant\_46;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Stage II)

//Compare the two aligned mantissas and determine which is smaller among both

if (S\_expo\_mant\_pipe1\_46 < L\_expo\_mant\_pipe1\_46)

begin

S\_mant\_46 = S\_expo\_mant\_pipe1\_46;

L\_mant\_46 = L\_expo\_mant\_pipe1\_46;

end

else

begin

S\_mant\_46 = L\_expo\_mant\_pipe1\_46;

L\_mant\_46 = S\_expo\_mant\_pipe1\_46;

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Add two mantissas if both num1 & num2 have same sign

// Subtract smaller mantissa from larger mantissa if num1 & num2 have different signs

if(e1\_pipe1\_46 != 0 & e2\_pipe1\_46 != 0)

begin

if(s1\_pipe1\_46 == s2\_pipe1\_46)

add\_mant\_46 = L\_mant\_46 + S\_mant\_46;

else

add\_mant\_46 = L\_mant\_46 - S\_mant\_46;

end

else

add\_mant\_46 = L\_mant\_46;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Stage III)

// Determine the amount of shift required to denormalise the mantisas &

// the corresponding direction (right or left shift)

if(add\_mant\_pipe2\_46[10])

begin

denorm\_shift\_46 = 4'd1;

denorm\_expo\_46 = 4'd1;

end

else if (add\_mant\_pipe2\_46[9])

begin

denorm\_shift\_46 = 4'd2;

denorm\_expo\_46 = 0;

end

else if (add\_mant\_pipe2\_46[8])

begin

denorm\_shift\_46 = 4'd3;

denorm\_expo\_46 = -1;

end

else if (add\_mant\_pipe2\_46[7])

begin

denorm\_shift\_46 = 4'd4;

denorm\_expo\_46 = -2;

end

else if (add\_mant\_pipe2\_46[6])

begin

denorm\_shift\_46 = 4'd5;

denorm\_expo\_46 = -3;

end

else

begin

denorm\_expo\_46 = 0;

end

sum\_expo\_46 = larger\_expo\_pipe3\_46 + denorm\_expo\_pipe3\_46;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Stage IV)

// Calculate sign bit, exponent and mantissa for final addition

if(denorm\_shift\_pipe3\_46 != 0)

add1\_mant\_46 = add\_mant\_pipe3\_46 << denorm\_shift\_pipe3\_46;

else

add1\_mant\_46 = add\_mant\_pipe3\_46;

sum\_mant\_46 = add1\_mant\_46[10:1];

if (s1\_pipe3\_46 == s2\_pipe3\_46)

sum\_sign\_46 = s1\_pipe3\_46;

if(e1\_pipe3\_46 > e2\_pipe3\_46)

sum\_sign\_46 = s1\_pipe3\_46;

else if(e2\_pipe3\_46 > e1\_pipe3\_46)

sum\_sign\_46 = s2\_pipe3\_46;

else

begin

if(m1\_pipe3\_46 > m2\_pipe3\_46)

sum\_sign\_46 = s1\_pipe3\_46;

else

sum\_sign\_46 = s2\_pipe3\_46;

end

sum\_46 = {sum\_sign\_46,sum\_expo\_46,sum\_mant\_46};

end

always @(posedge clk)

begin

if(reset)

begin

// Reg\_1 Output

larger\_expo\_pipe1\_46 <= 0;

S\_expo\_mant\_pipe1\_46 <= 0;

L\_expo\_mant\_pipe1\_46 <= 0;

s1\_pipe1\_46 <= 0;

s2\_pipe1\_46 <= 0;

e1\_pipe1\_46 <= 0;

e2\_pipe1\_46 <= 0;

m1\_pipe1\_46 <= 0;

m2\_pipe1\_46 <= 0;

// Reg\_2 Output

add\_mant\_pipe2\_46 <= 0;

larger\_expo\_pipe2\_46 <= 0;

s1\_pipe2\_46 <= 0;

s2\_pipe2\_46 <= 0;

e1\_pipe2\_46 <= 0;

e2\_pipe2\_46 <= 0;

m1\_pipe2\_46 <= 0;

m2\_pipe2\_46 <= 0;

// Reg\_3 Output

denorm\_expo\_pipe3\_46 <= 0;

denorm\_shift\_pipe3\_46 <= 0;

larger\_expo\_pipe3\_46 <= 0;

add\_mant\_pipe3\_46 <= 0;

s1\_pipe3\_46 <= 0;

s2\_pipe3\_46 <= 0;

e1\_pipe3\_46 <= 0;

e2\_pipe3\_46 <= 0;

m1\_pipe3\_46 <= 0;

m2\_pipe3\_46 <= 0;

sum\_46 <= 0;

end

else

begin

///////////////////////// Pipelining stages /////////////////////////

// Propagate the variable to the next stgae through register

larger\_expo\_pipe1\_46 <= larger\_expo\_46;

S\_expo\_mant\_pipe1\_46 <= S\_expo\_mant\_46;

L\_expo\_mant\_pipe1\_46 <= L\_expo\_mant\_46;

s1\_pipe1\_46 <= s1\_46;

s2\_pipe1\_46 <= s2\_46;

e1\_pipe1\_46 <= e1\_46;

e2\_pipe1\_46 <= e2\_46;

m1\_pipe1\_46 <= m1\_46;

m2\_pipe1\_46 <= m2\_46;

add\_mant\_pipe2\_46 <= add\_mant\_46;

larger\_expo\_pipe2\_46 <= larger\_expo\_pipe1\_46;

s1\_pipe2\_46 <= s1\_pipe1\_46;

s2\_pipe2\_46 <= s2\_pipe1\_46;

e1\_pipe2\_46 <= e1\_pipe1\_46;

e2\_pipe2\_46 <= e2\_pipe1\_46;

m1\_pipe2\_46 <= m1\_pipe1\_46;

m2\_pipe2\_46 <= m2\_pipe1\_46;

denorm\_expo\_pipe3\_46 <= denorm\_expo\_46;

denorm\_shift\_pipe3\_46 <= denorm\_shift\_46;

larger\_expo\_pipe3\_46 <= larger\_expo\_pipe2\_46;

add\_mant\_pipe3\_46 <= add\_mant\_pipe2\_46;

s1\_pipe3\_46 <= s1\_pipe2\_46;

s2\_pipe3\_46 <= s2\_pipe2\_46;

e1\_pipe3\_46 <= e1\_pipe2\_46;

e2\_pipe3\_46 <= e2\_pipe2\_46;

m1\_pipe3\_46 <= m1\_pipe2\_46;

m2\_pipe3\_46 <= m2\_pipe2\_46;

end

end

endmodule

**3-stage Pipelined Half-precision Floating Point Adder Testbench:**

timescale 1ns/10ps

module HP\_FP\_Pipelined\_Adder\_Test();

reg [15:0] num1\_46,num2\_46;

reg reset\_46,clk\_46;

wire [15:0] Sum\_46;

HP\_FP\_Pipelined\_Adder a1(clk\_46,reset\_46,num1\_46,num2\_46,Sum\_46);

always

#2 clk\_46 = ~clk\_46;

initial

begin

clk\_46 = 1;

reset\_46 = 1;

#1 reset\_46 = 0;

end

initial

begin

num1\_46 = 16'h5630; //99

num2\_46 = 16'hD590; //-89

// Sum\_46 = 16'h4900; //10

#4 num1\_46 = 16'hDC6C; //-283

num2\_46 = 16'hD420; //-66

// Sum\_46 = 16'hDD74; //-349

#4 num1\_46 = 16'h5948; //169

num2\_46 = 16'hD948 ; //-169

// Sum\_46 = 16'h0000; //0

#30 $finish;

end

initial

begin

$dumpfile("HP\_FP\_Pipelined\_Adder.vcd");

$dumpvars(0);

$display(" Num\_1 | Num\_2 | Sum |");

$monitor($time," %h | %h | %h |",num1\_46,num2\_46,Sum\_46);

end

endmodule

**4x4 Matrix keypad:**

module keypad (clk,row,col,seg1,seg2,seg3,seg4,rst,s1,s2,s3,sw,enable,done,count);

input enable;

output done;

input [3:0] sw;

input rst,s1,s2,s3;

input clk;

input [3:0] col;

output [3:0] row;

output reg [7:0] seg1;

output reg [7:0] seg2;

output reg [7:0] seg3;

output reg [7:0] seg4;

output reg [1:0] count;

wire [7:0] sego\_2,sego\_3,sego\_4,sego\_1;

wire [7:0] seg;

wire [3:0] decoded\_in;

wire [3:0] decoded\_mux;

wire [15:0] numA;

wire [15:0] numB;

wire [15:0] numC;

wire sel;

wire enable,done;

reg [15:0] tmpA;

reg [15:0] tmpB;

reg [3:0] decoded\_out;

wire [15:0]sumC;

assign numA = tmpA;

assign numB = tmpB;

assign sel = 0;

display D0(decoded1,sego\_1);

display D3(decoded2,sego\_2);

display D4(decoded3,sego\_3);

display D5(decoded4,sego\_4);

display D (decoded\_mux,seg);

seg7 S (clk, row, col ,decoded\_in);

HP\_FP\_Pipelined\_Adder F1(clk,numA,numB,enable,done,numC);

mux M1 (decoded\_in,decoded\_out,sel,decoded\_mux);

reg [3:0] decoded1,decoded2,decoded3,decoded4;

always @ (posedge clk)

begin

if({rst,s1,s2,s3} == 4'b1000)

begin

count = 2'b00;

{seg4,seg3,seg2,seg1} = {4{8'b11000000}};

tmpA = 0;

tmpB = 0;

end

if({rst,s1,s2,s3} == 4'b0100)

begin

count = 2'b01;

if (sw==4'b0001) begin

seg1 = seg;

tmpA[3:0] = decoded\_in;

end

if (sw == 4'b0010) begin

seg2 = seg;

tmpA[7:4] = decoded\_in;

end

if (sw == 4'b0100) begin

seg3 = seg;

tmpA[11:8] = decoded\_in;

end

if (sw == 4'b1000) begin

seg4 = seg;

tmpA[15:12] = decoded\_in;

end

end

if({rst,s1,s2,s3} == 4'b0010)

begin

count = 2'b10;

if (sw==4'b0001) begin

seg1 = seg;

tmpB[3:0] = decoded\_in;

end

if (sw == 4'b0010) begin

seg2 = seg;

tmpB[7:4] = decoded\_in;

end

if (sw == 4'b0100) begin

seg3 = seg;

tmpB[11:8] = decoded\_in;

end

if (sw == 4'b1000) begin

seg4 = seg;

tmpB[15:12] = decoded\_in;

end

end

if({rst,s1,s2,s3} == 4'b0001)

begin

count = 2'b11;

decoded1 = numC[3:0];

seg1 = sego\_1;

decoded2 = numC[7:4];

seg2 = sego\_2;

decoded3 = numC[11:8];

seg3 = sego\_3;

decoded4 = numC[15:12];

seg4 = sego\_4;

end

end

endmodule

**7-segment decode Logic:**

module display (DisVal, SegOut);

input [3:0] DisVal;

output [7:0] SegOut;

reg [7:0] SegOut;

always @ (DisVal)

begin

case(DisVal)

4'h0 : SegOut = 8'b11000000; // 0

4'h1 : SegOut = 8'b11111001; // 1

4'h2 : SegOut = 8'b10100100; // 2

4'h3 : SegOut = 8'b10110000; // 3

4'h4 : SegOut = 8'b10011001; // 4

4'h5 : SegOut = 8'b10010010; // 5

4'h6 : SegOut = 8'b10000010; // 6

4'h7 : SegOut = 8'b11111000; // 7

4'h8 : SegOut = 8'b10000000; // 8

4'h9 : SegOut = 8'b10010000; // 9

4'hA : SegOut = 8'b10001000; // A

4'hB : SegOut = 8'b10000011; // B

4'hC : SegOut = 8'b11000110; // C

4'hD : SegOut = 8'b10100001; // D

4'hE : SegOut = 8'b10000110; // E

4'hF : SegOut = 8'b10001110; // F

default : SegOut = 8'b10111111;

endcase

end

endmodule

**Keypad scan Logic:**

module seg7 (clk, row , col, dec);

input clk;

input [3:0] col;

output [3:0] row;

output [3:0] dec;

reg [3:0] row;

reg [3:0] dec;

reg [29:0] delay;

always @ (posedge clk)

begin

delay <= delay + 1'b1;

case(delay[20:18])

3'b000 : row <= 4'b0111;

3'b001 :

begin

if (col == 4'b0111)

dec <= 4'h1; // 1

if (col == 4'b1011)

dec <= 4'h2; // 2

if (col == 4'b1101)

dec <= 4'h3; // 3

if(col == 4'b1110)

dec <= 4'hA; //A

end

3'b010 : row <= 4'b1011;

3'b011 :

begin

if (col == 4'b0111)

dec <= 4'h4; // 4

if (col == 4'b1011)

dec <= 4'h5; // 5

if (col == 4'b1101)

dec <= 4'h6; // 6

if (col == 4'b1110)

dec <= 4'hB; // B

end

3'b100 : row <= 4'b1101;

3'b101 :

begin

if (col == 4'b0111)

dec <= 4'h7; // 7

if (col == 4'b1011)

dec <= 4'h8; // 8

if (col == 4'b1101)

dec <= 4'h9; // 9

if (col == 4'b1110)

dec <= 4'hC; // C

end

3'b110 : row <= 4'b1110;

3'b111 :

begin

if (col == 4'b0111)

dec <= 4'hF; // F

if (col == 4'b1011)

dec <= 4'h0; // 0

if (col == 4'b1101)

dec <= 4'hE; // E

if (col == 4'b1110)

dec <= 4'hD; // D

end

endcase

end

endmodule

**Output Multiplex logic:**

module mux (a,b,sel,c);

input [3:0] a,b;

input sel;

output [3:0] c;

assign c = sel ? b : a;

endmodule