Floating Point Adder

EE18ACMTECH11006 BHAVANI MACHNOORI
EE18MTECH11005 AJAY HASE

April 26, 2019

1 Introduction

Floating-point addition is a fundamental component of math coprocessors, dsp processors, embedded arithmetic processors, and data processing units. The dramatic increase in application size has allowed FPGAs to be considered for several scientific applications that require floating-point arithmetic. The advantage of floating-point arithmetic over fixed-point arithmetic is the range of numbers that can be represented with the same number of bits.

On the other hand, floating point operations usually are slightly slower than integer operations, and you can lose precision.

1.Floating Point Number

The term floating point is derived from the meaning that there is no fixed number of digits before and after the decimal point, that is, the decimal point can float. There was also a representation in which the number of digits before and after the decimal point is set, called fixed-point representations. In general floating point, representations are slower and less accurate than fixed-point representations, but they can handle a larger range of numbers. Floating Point Numbers are numbers that consist of a fractional part. For e.g. following numbers are the floating point numbers: 35, -112.5, , 4E-5 etc. Floating point arithmetic is considered a tough subject by many peoples. This is rather surprising because floating-point is found in computer systems. Almost every language supports a floating point data type. A number representation (called a numeral system in mathematics) specifies some way of storing a number that maybe encoded as a string of digits. In computing, floating point describes a system for numerical representation in which a string of digits (or bits) represents a rational number. The term floating point refers to the fact that the radix point (decimal point, or more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number.

2. Floating Point representation

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point wcomputationhich was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PCs, Macs, and most Unix platforms. There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

i) The Sign of Mantissa :
This is as simple as the name. 0 represents a positive number while 1 represents a negative number.

ii) The Biased exponent :
The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

iii) The Normalised Mantisa :
The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

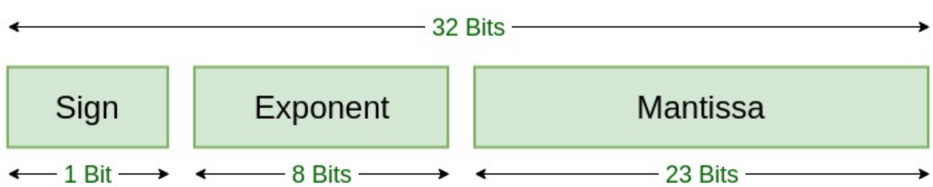

Figure 1: Single precision format

2 Floating Point Adder

We use a simplified 10-bit format in this example and ignore the round-off error. The representation consists of a 3-bit exponent field, e, which represents the exponent; and a 7-bit significand field, f, which represents the significant or the fraction. In this format, the value of a floating-point number is .f * 2^e. The . f * 2^e is the magnitude of the number. The floating-point representation can be considered as a variation of the sign-magnitude format. We also make the following assumptions:

i) Both exponent and significand fields are in unsigned format. The representation has to be either normalized or zero.

ii) Normalized representation means that the MSB of the significand field must be1. If the magnitude of the computation result is smaller than the smallest normalized nonzero magnitude, 0.10000000 * 2^{0000}, it must be converted to zero.

In order to pre-normalize o pre-normalize the mantissa of the number with the smaller exponent, a right-shifter is used to right-shift the mantissa by the absolute-exponent difference. This is done so that the two numbers will have the same exponent and normal integer addition can be carried out. The right-shifter is one of the most important modules to consider when designing for latency, as it adds significant delay.

Block Diagram:

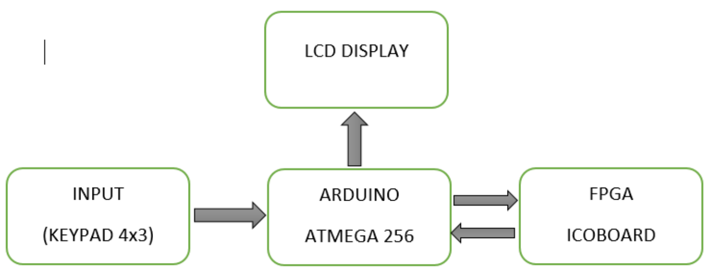

Figure 2: Blockdiagram

The input of two numbers is taken from keypad which is interface with arduino

board. Arduino convert the both the numbers into floating point format of 3 exponent bits and 7 mantissa bits. The converted floating point binary numbers are parallely transmitted to icoboard. The floating point addition operation is performed by icoboard describe as below in the flowchart.And completing addition the floating point output is fed back to arduino. Finally arduino converts the floating point binary represented number into decimal fraction number and display it on LCD display.
The computation is done in four major steps:

1. Sorting : Puts the number with the larger magnitude on the top and the number with the smaller magnitude on the bottom (we call the sorted numbers "big number" and "small number").

2. Alignment: Aligns the two numbers so that they have the same exponent. This can be done by adjusting the exponent of the small number to match the exponent of the big number. The significant of the small number has to shift to the right according to the difference in exponents.

3. Addition: adds the significands of two aligned numbers.

4. Normalization: adjusts the result to the normalized format. Three types of normalization procedures may be needed:
i. The result may contain leading zeros in front.
ii. The result may be too small to be normalized and thus needs to be converted to zero.
iii.The result may generate a carry-out bit.

Flow Chart of Floating Point Addition:

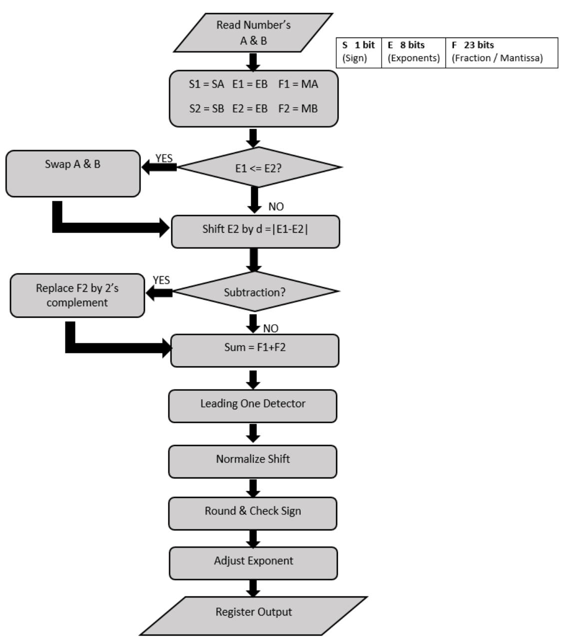

Figure 3: Flowchart

Arduino Code:

```
#include <LiquidCrystal.h>

#include <Keypad.h>

LiquidCrystal lcd(29,30,31,32,33,34);//RS,EN,D4,D5,D6,D7

const byte ROWS = 4; //four rows
const byte COLS = 3; //three columns
char keys[ROWS][COLS] = {
  {'1','2','3'},
  {'4','5','6'},
  {'7','8','9'},
  {'.','0','#'}
};

byte rowPins[ROWS] = {25,23,22,28};
byte colPins[COLS] = { 24,26,27 };
Keypad kpd = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS
    );


void setup(){
  for(int k=29;k<35;k++){
      pinMode(k,OUTPUT);
    }
  for(int k=0;k<8;k++){
      pinMode(k,OUTPUT);
    }
  for(int k=36;k<51;k = k+2){
      pinMode(k,OUTPUT);
    }
  for(int k=39;k<54;k = k+2){
      pinMode(k,INPUT);
    }
      pinMode(13,OUTPUT);
      pinMode(37,OUTPUT);
      pinMode(52,INPUT);
      pinMode(8,INPUT);
      pinMode(9,OUTPUT);
      pinMode(10,OUTPUT);
      pinMode(11,OUTPUT);
      pinMode(12,OUTPUT);
      lcd.begin(16, 2);//initializing LC
      pinMode(35, HIGH);
      digitalWrite(35,HIGH);

  Serial.begin(9600);
}

void loop()
{
   float v1 = 0;
   float v2 = 0;
   int num1= 0;
   int num2 = 0;
```

```
  boolean Bin1[] = {0,0,0,0,0,0,0,0,0,0};
  boolean Bin2[] = {0,0,0,0,0,0,0,0,0,0};
  boolean Binfinal[] = {0,0,0,0,0,0,0,0,0,0};
  boolean expOut1[]  = {0,0,0};
  boolean expOut2[]  = {0,0,0};
  boolean binOut1[] = {0,0,0,0,0,0,0};
  boolean binOut2[] = {0,0,0,0,0,0,0};

  boolean decOut[] = {0,0,0,0};
  boolean fracOut[] = {0,0,0,0};
  float addition = 0.0;

  Serial.print("Enter NO 1 = ");
  lcd.print("Enter NO 1 = ");
  lcd.setCursor(0,1);
  delay(1000);

    fun(&v1);

    lcd.print(v1);
    Serial.parseFloat();
    Serial.println(v1);

    delay(3000);


  lcd.clear();
  lcd.print("Enter NO 2 = ");
  Serial.print("Enter NO 2 = ");
    delay(1000);

    fun(&v2);

    lcd.setCursor(0,1);
    lcd.print(v2);
    delay(3000);
    Serial.parseFloat();
    Serial.println(v2);
lcd.clear();
    delay(1000);

  convertDecToBin(v1,Bin1);
  binaryToFloating(Bin1,binOut1,expOut1);
  delay(1000);

  convertDecToBin(v2,Bin2);
  binaryToFloating(Bin2,binOut2,expOut2);
  delay(1000);

  Serial.print("floating point representation of numbers are ");
  Serial.print("\n");
  for(int i =0; i<7;i++){
    Serial.println(binOut1[i]);
    Serial.print("\t");
    Serial.println(binOut2[i]);
    delay(100);
```

```
    }
    Serial.print("exponents  are");
    Serial.print("\n");
    for(int i=0; i<3;i++){
      Serial.println(expOut1[i]);
      Serial.print("\t");
      Serial.println(expOut2[i]);
      delay(100);
    }

  digitalWrite(5,expOut1[2]);
  digitalWrite(2,expOut1[1]);
  digitalWrite(50,expOut1[0]);
  digitalWrite(48,binOut1[6]);
  digitalWrite(46,binOut1[5]);
  digitalWrite(44,binOut1[4]);
  digitalWrite(42,binOut1[3]);
  digitalWrite(40,binOut1[2]);
  digitalWrite(38,binOut1[1]);
  digitalWrite(36,binOut1[0]);

  digitalWrite(6,expOut2[2]);
  digitalWrite(3,expOut2[1]);
  digitalWrite(12,expOut2[0]);
  digitalWrite(11,binOut2[6]);
  digitalWrite(10,binOut2[5]);
  digitalWrite(9,binOut2[4]);
  digitalWrite(13,binOut2[3]);
  digitalWrite(37,binOut2[2]);
digitalWrite(7,binOut2[1]);
  digitalWrite(4,binOut2[0]);
  delay(15000);

  Binfinal[0] = digitalRead(39);
  Binfinal[1] = digitalRead(41);
  Binfinal[2] = digitalRead(43);
  Binfinal[3] = digitalRead(45);
  Binfinal[4] = digitalRead(47);
  Binfinal[5] = digitalRead(49);
  Binfinal[6] = digitalRead(51);
  Binfinal[7] = digitalRead(53);
  Binfinal[8] = digitalRead(8);
  Binfinal[9] = digitalRead(52);



 Serial.print("addition is");
 Serial.print("\n");
 for(int i=0; i<10;i++){
     Serial.println(Binfinal[i]);
     delay(100);
    }


    floatToDec(Binfinal,decOut,fracOut);
    addition = decimalFraction(decOut,fracOut);
    addition = addition + 765.0;
```

```
  lcd.print("addition = ");
  lcd.setCursor(0,1);
  lcd.print(addition);
  Serial.println(addition);
  delay(10000);
  lcd.clear();

}
//
    ****************************************************************************************

void convertDecToBin(float Dec, boolean Bin[]) {
  int decimal = int(Dec);
  float fraction = Dec-decimal;
  for(int i =0 ; i <= 4 ; i++) {
      int rem  = decimal%2;
      Bin[5+i] = rem;
decimal = int(decimal/2);
  }
  for(int i =0;i<=4;i++){
    fraction = fraction * 2;
    Bin[4-i] = int(fraction);
    fraction = fraction - int(fraction);
  }
}
//
    ######################################################################################################################################################################

void binaryToFloating( boolean Bin[],boolean binOut[],boolean
    expOut[]){

  if(Bin[9]==1){expOut[2]=1;expOut[1]=0;expOut[0]=1;
  for(int i=0;i<=6;i++){
  binOut[i] =Bin[3+i];}
  }
  else if(Bin[8]==1){expOut[2]=1;expOut[1]=0;expOut[0]=0;
  for(int i=0;i<=6;i++){
  binOut[i] =Bin[2+i];}}

  else if(Bin[7]==1){
   expOut[2]=0;expOut[1]=1;expOut[0]=1;
  for(int i=0;i<=6;i++){
  binOut[i] =Bin[1+i];}}

  else if(Bin[6] ==1){
    expOut[2]=0;expOut[1]=1;expOut[0]=0;
   for(int i=0;i<=6;i++){
  binOut[i] =Bin[i];}}

  else {
    expOut[2]=0;expOut[1]=0;expOut[0]=1;
     for(int i=6;i>=1;i--){
    binOut[i] =Bin[i-1];}
    }
```

```

}

//###################################################################3
void floatToDec(boolean Binfinal[],boolean decOut[],boolean fracOut[]){

   if(Binfinal[9]==0 && Binfinal[8]==0 && Binfinal[7]==0){

     fracOut[3]=Binfinal[6];fracOut[2]=Binfinal[5];fracOut[1]=Binfinal[4];fracOut[0]=Binfinal[3];
 }
  else if(Binfinal[9]==0 && Binfinal[8]==0 && Binfinal[7]==1){
    decOut[0] = Binfinal[6];
    fracOut[3]=Binfinal[5];fracOut[2]=Binfinal[4];fracOut[1]=Binfinal[3];fracOut[0]=Binfinal[2];
  }
  else if(Binfinal[9]==0 && Binfinal[8]==1 && Binfinal[7]==0){
    decOut[0] = Binfinal[5];   decOut[1]= Binfinal[6];
    fracOut[3]=Binfinal[4];fracOut[2]=Binfinal[3];fracOut[1]=Binfinal[2];fracOut[0]=Binfinal[1];
    }
  else if(Binfinal[9]==0 && Binfinal[8]==1 && Binfinal[7]==1){
    decOut[0] = Binfinal[4]; decOut[1]=Binfinal[5]; decOut[2]=Binfinal[6];
    fracOut[3]=Binfinal[3]; fracOut[2]=Binfinal[2]; fracOut[1]=Binfinal[1]; fracOut[0]=Binfinal[0];
  }
  else if(Binfinal[9]== 1 && Binfinal[8]==0 && Binfinal[7]==0){
    decOut[0]= Binfinal[3]; decOut[1]= Binfinal[4]; decOut[2]=Binfinal[5];  decOut[3] =Binfinal[6];
    fracOut[3]=Binfinal[2];fracOut[2]=Binfinal[1];fracOut[1]=Binfinal[0];fracOut[0] =0;
  }
  else{
    decOut[0] = Binfinal[3]; decOut[1] =Binfinal[4]; decOut[2]=Binfinal[5]; decOut[3] =Binfinal[6];
    fracOut[3]=Binfinal[2]; fracOut[2]=Binfinal[1]; fracOut[1]=Binfinal[0];fracOut[0]=0;

  }
}
//#########################################################################
float decimalFraction(boolean decOut[],boolean fracOut[])
 {
  float intDecimal = 0;
  float fracDecimal = 0;
  float tw = 1.0;
  float add = 0.0;

  for(int i=0;i<=3;i++){
   intDecimal = float(intDecimal) + float((int(decOut[i]) -'0')*tw);
  tw = tw*2.0;
}
```

```
float twos = 2.0;
for(int i=3;i>=0;i--){
  fracDecimal = fracDecimal + float((fracOut[i]-'0')/twos);
  twos *=2.0;
}
add = float(intDecimal) + fracDecimal;

return add;

}

//################################################################################

 void fun(float *x){


   int  num = 0;
   int num2 = 0;
   int a = 0;
   float z = 0.0;
   float number = 0.0;
   char key = kpd.getKey();
   while(key != '#')
   {
      switch (key)
      {
         case NO_KEY:
            break;

         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':
            a = a + 1;
            num = num * 10 + (key - '0');

            break;

         case '.':
            num2 = num;
            num = 0;
            a = 0;

            break;
      }

      key = kpd.getKey();
   }
   z = pow(10,a);
   number = num2 + float(num /z);
   *x = number;
 }
```

Verilog code for Icoboard:

```
module fp_adder
(
input  clk,
input [2:0] exp1, input [2:0] exp2,
input [6:0] frac1, input [6:0] frac2,

output  reg [2:0] exp_out ,
output  reg [6:0] frac_out
);

reg [2:0] expb , exps , expn , exp_diff ;
reg [6:0] fracb , fracs , fraca , fracn , sum_norm ;
reg [7:0] sum;
reg [2:0] lead0;
reg [7:0] num1 , num2;
reg [26:0] delay;
reg t;
reg [31:0] delta;

initial begin
expb <= 3'b000;
exps <= 3'b000;
expn <= 3'b000;
exp_diff <= 3'b000;
fracb <= 7'd0;
fracs <= 7'd0;
fraca <= 7'd0;
fracn <= 7'd0;
sum_norm <= 7'd0;
sum <= 8'd0;
lead0 <= 3'd0;
num1 <=8'd0;
num2 <= 8'd0;
delta <=32'd0;

end

// body
always@ (posedge clk)
begin
// 1st stage: sort to find the larger number
delay <= delay + 27'd1;

if (delay == 27'b000011101011110000100000000)
```

```
    begin
    delay <= 27'd0;
if(frac2 != 7'b00000000)
begin
if({exp1,frac1}>{exp2,frac2})
        begin
        expb <= exp1;
        exps <= exp2;
        fracb <= frac1;
        fracs <= frac2;
        end
    else
        begin
        expb<=exp2;
        exps<=exp1;
        fracb<=frac2;
        fracs<=frac1;
        end

// 2nd stage: align smaller number
    exp_diff <= expb - exps;
    if(exp_diff == 3'b000)
        begin
        fraca <= fracs >> 0;
        end
    else if(exp_diff == 3'b001)
        begin
        fraca <= fracs >> 1;
        end
    else if(exp_diff == 3'b010)
        begin
        fraca <= fracs >> 2;
        end
    else
        begin
        fraca <= fracs >> 3;
        end

    num1[6:0] <= fracb;
    num2[6:0] <= fraca;
    // 3rd stage:addition
    sum <= num1 + num2;

// 4th stage: normalize
// count leading 0s
if (sum[6]==1'b1)
```

```
begin
  sum_norm <= sum << 0;
   lead0 <= 3'o0;
end

   else if (sum[5]==1'b1)
       begin
       sum_norm <= sum << 1;
       lead0 <= 3'o1;
       end

   else if (sum[4]==1'b1)
       begin
       sum_norm <= sum << 2;
       lead0 <= 3'o2;
       end

   else if (sum[3]==1'b1)
       begin
       sum_norm <= sum << 3;
       lead0 <= 3'o3;
       end

   else if (sum[2]==1'b1)
       begin
       sum_norm <= sum << 4;
       lead0 <= 3'o4;
       end

   else if (sum[1]==1'b1)
       begin
       sum_norm <= sum << 5;
       lead0 <= 3'o5;
       end

   else
       begin
       sum_norm <= sum << 6;
       lead0 <= 3'o6;

   end



//here
if(sum[7]==1)
```

```
    begin
    expn <=expb+1;
    fracn <=sum[7:1];
    end
else if (lead0 > expb) // too small to normalize
    begin
    expn <= 0; // set to 0
    fracn <= 0;
    end
else
    begin
    expn <= expb - lead0;

    fracn <= sum_norm;
end

exp_out <= expn;
frac_out <= fracn;
t<=1;
if(t==1)
begin
delta <= delta + 32'd1;
if (delta == 32'b01000101111101011110000100000000)
    begin
    delta <= 32'd0;
    t<= 0;
    end
    end
    end



end
end
endmodule
```

.pcf File for above code is

```
set_io clk R9

set_io frac1[0] A5
set_io frac1[1] A2
set_io frac1[2] C3
```

```
set_io frac1[3] B4
set_io frac1[4] B7
set_io frac1[5] B6
set_io frac1[6] B3
set_io exp1[0] B5
set_io exp1[1] M8
set_io exp1[2] L7

set_io frac2[0] T9
set_io frac2[1] T10
set_io frac2[2] T13
set_io frac2[3] R14
set_io frac2[4] R10
set_io frac2[5] T11
set_io frac2[6] T14
set_io exp2[0] T15
set_io exp2[1] P9
set_io exp2[2] G5

set_io frac_out[0] D8
set_io frac_out[1] B9
set_io frac_out[2] B10
set_io frac_out[3] B11
set_io frac_out[4] B8
set_io frac_out[5] N6
set_io frac_out[6] A10
set_io exp_out[0] A11
set_io exp_out[1] L9
set_io exp_out[2] N9
```

Make file:

```
 .PHONY: default
default: prog_sram

$(v_fname).blif: $(v_fname).v
        yosys -p 'synth_ice40 -blif $(v_fname).blif' $(v_fname).v

$(v_fname).asc: $(v_fname).blif $(v_fname).pcf
        arachne-pnr -d 8k -p $(v_fname).pcf -o $(v_fname).asc $(v_fname).blif

$(v_fname).bin: $(v_fname).asc
        icetime -d hx8k -c 25 $(v_fname).asc
        icepack $(v_fname).asc $(v_fname).bin
```

```
prog_sram: $(v_fname).bin
	icoprog -p < $(v_fname).bin
```

CONCLUSION:

A single precision floating-point adder is implemented in this paper. Floating point fpga arithmetic unit are useful for single precision and double precision and quad precision .This precision specify various bit of operation. This is invaluable tools in the implementation of high performance systems, combining the reprogrammability advantage of general purpose processors with the speed and parallel processing. A general purpose arithmetic unit require for all operations. In this paper single precision floating point addition arithmetic calculation is described. Fpga implementation of Floating point arithmetic calculation provide various step which are require for calculation. Normalization and alignment are useful for operation and floating point number should be normalized before any calculation.