Project Report

In the Partial fulfillment of B. Tech. - III year

Course requirement of

Subject: Digital Hardware Design

BML MUNJAL UNIVERSITY™
FROM HERE TO THE WORLD

Submitted To:

Dr. Nirupama MP
Assistant Professor (Electronics and Communication Engineering)
School of Engineering & Technology
BML Munjal University

Submitted by:

S No.	Roll No.	Name
1.	1700315C204	Harsha Machineni
2.	1700316C204	Harshith Kumar.N
3.	1700327C204	Madhur Garg
4.	1700361C204	Sushrut Gokhale
5.	1700346C204	Saikumar Pagidipalli
6.	1700340C204	Phani Sriram Maganti

Date: 26/11/2019
Branch: ECE- 2017

Table of Contents

Abstract

Floating Point (FP) multiplication is widely used in large set of scientific and signal processing computation. Multiplication is one of the common arithmetic operations in these computations. In addition, the proposed design is compliant with IEEE-754 format and handles various special cases. The design achieved with an area of slices.

Problem statement

State the problem or problems that motivated or required a solution provided by this project: Floating point multipliers play an important role in calculations in digital world. No processing can be imagined without them. Because of the complexity of the algorithms, floating point operations are very hard to implement on FPGA. The computations for floating point operations involve large dynamic range, but the resources required for this operation is high compared with the integer operations. We have unsigned/signed multiplier for multiplication of binary numbers, for multiplication of floating-point numbers floating point multiplier is used. There is a separate algorithm for this multiplication.

Specific Problem Solved

We mainly focus on the fast multiplication of floating-point numbers using less no. of flip flops.

Project in Detail

.

1. Floating Point multiplication algorithm:

Multiplying two numbers in floating point format is done by

1. Adding the exponent of the two numbers then subtracting the bias from their result.
2. Multiplying the significand of the two numbers
3. Calculating the sign by XORing the sign of the two numbers.

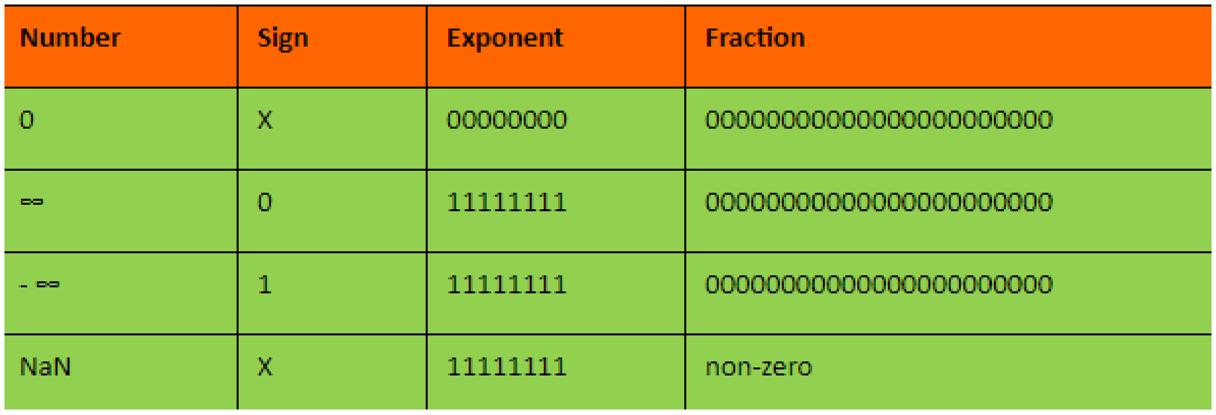
In order to represent the multiplication, result as a normalized number there should be ‘1’ in the MSB of the result (leading one).

2. The following steps are necessary to multiply two floating point numbers:

1. Multiplying the significand i.e. (1.MI * 1.M2).
2. Placing the decimal point in the result.
3. Adding the exponents i.e. (E I + E2 -Bias).
4. Obtaining the sign i.e. sign_1 XOR sign_2.
5. Normalizing the result i.e. obtaining '1' at the MSB of the results "significand".

3. Working of Multiplier:

- Initially the state is Reset when reset is '1' the input & output_ack and input & output_stb bits are '0' , irrespective of inputs when the reset is '0'. Then the next state is "get_a".
- In get_a, if both input_ack and input_stb are '1', then the given first input is stored in respective 64-bit register, and next state is to get_b.
- In get_b, if both input_ack and input_stb are '1', then the given first input is stored in respective 64-bit register, and next state is to unpack.
- In this state the 64-bit registers are unpacked into sign bit, Exponent, mantissa. Since, we are using IEEE-754 Double precision Floating point multiplier. The exponent field contains 11 bits and Mantissa field contains 52 bits and 1 bit for sign.
- After, Unpacking the given inputs then we need to check for special cases, here, we considered the following:
    1. Quiet NaN
    2. Infinity
    3. Zero
    4. DE normalised



Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

- A number is said to be zero if the exponent field and mantissa field is filled with all zeros.
- A number is said to be infinity if the exponent is filled with all ones and mantissa is filled with all zeros. As per sign defines the Infinity could be -infinity and +infinity.
- A number is said to be NaN(not a number) if the exponent field is filled with all ones and the mantissa with non-zero. If the 52th bit is '1' then it is said to be Quiet NaN otherwise it is said to be Single NaN.
- After checking special cases, the next state will be Normalisation of inputs as per IEEE-754 format i.e. $\mathbf{(-1)^{S}\,x(1.M)\,x2^{(E\text{-}Bias)}}$. Then the next state will be multiply.
- Here, in this state we are going to xor the sign bits and Adding exponents and Multiplying the mantissas and store them in temporary registers.
- Then Normalising the mantissa and going for packing the components.
- After packing the sign, Exponent and Mantissa the Next state is Put the package in 64-bit output register.

4. Result:

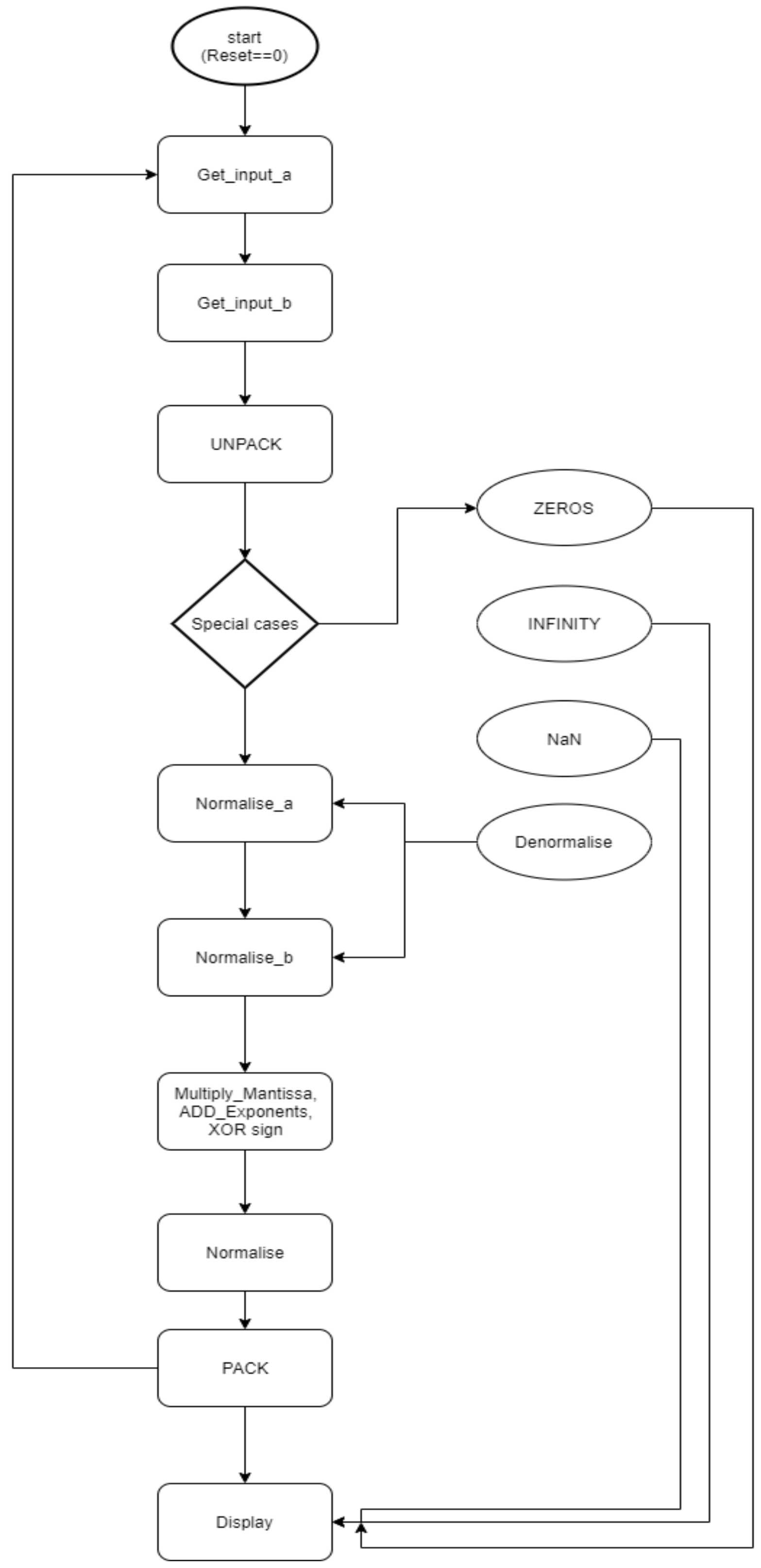
Logic Utilization	**used**	**Available**	**Util%**
Number of slice registers (Flip-flops)	**401**	**41000**	**0.97**
Number of LUT's	**504**	**82000**	**0.97**
Number of bonded IO's	**200**	**300**	**66.67**

Table1: Device utilization summary of IEEE-754 Double precision floating point multiplier

Previous Experiments

S.No.	Existing state of art	Drawbacks in existing state of art	Overcome
1.	An FPGA Based High Speed IEEE-754 Double Precision Floating Point Multiplier U sing Verilog	It is using a greater number of flip-flops; hence hardware requirement is more.	In our implementation we have tried to reduce the number of flip-flops and use minimum hardware possible.
2.	An TEEE-754 single precision pipelined floating point multiplier is implemented on multiple FPGAs (4 Actel AI280). Nabeel Shirazi, Walters, and Peter Athanasn implemented custom 16/18 bit three stage pipelined floating point multiplier	doesn't support rounding modes	Supports rounding modes.
3.	A parameterizable floating point multiplier is implemented using five stages pipeline, Handel-C software and Xilinx XCYIOOO FPGA.The design achieved the operating frequency of 28MFlops	The multiplier handles the overflow and underflow cases but rounding is not implemented.	Supports rounding modes.
4.	The floating point unit is implemented using the primitives of Xilinx Yirtex IT FPGA. The design achieved the operating frequency of 100 MHz with a latency of 4 clock cycles.	The multiplier handles the overflow and underflow cases but rounding is not implemented.	Supports rounding modes.

Block Diagram



Software Used

Xilinx Vivado 2018.2 and Modelsim for simulation.

Why it is better!

The circuit that is designed is able to multiply two single or double precision floating point numbers. The circuit used significantly less amount of flip flops as compares to other implementations on the market however it is able to multiply faster and with less error.

Project Status

The Project met objectives and RTL schematic generated successfully and Implementation completed.

Project Motivation

Multipliers form an important part of computing. It’s relatively easy to design a simple whole number multiplier than the floating-point multipliers. Although, there are various algorithms and fine functioning floating point multiplying hardware, but their power consumption and area coverage is more. By studying many such multipliers, the idea to reduce the hardware requirement, without compromising with the speed was conceived.

Product Implementation

Floating point Multiplication is very important in high power computing applications such as:

- Image signal processing
- Digital signal processing
- weather forecasting
- FFT’s Digital filters and various Transforms
- For computing very large number and very small number we will use these floating-point Multipliers.

Additional Information

As FPGA models and technology continue to evolve, floating-point hardware in the DSP blocks will gradually replace traditional fixed-point FFT implementations, with substantial savings of design time and FPGA LUT/register fabric.

Floating-point FFT with Minimal Hardware

There is a natural preference to use floating-point implementations in custom embedded applications because they offer a much higher dynamic range and as a bypass the design and analyzing fixed-point word-lengths.
Which includes

- How to properly “scale” operations to minimize word growth and avoid overflow.
- Additionally, many applications such as professional audio, industrial measurement/process control, imaging radar, signal intelligence and scientific computing inherently require a high dynamic range so that floating-point support is a necessity.

References

1. B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on VLS1, vol. 2, no. 3, pp. 365-367, 1994.
2. N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM"95), pp.155-162, 1995.
3. L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM"96), pp. 107-116,1996.

4. An FPGA Based High Speed IEEE-754 Double Precision Floating Point Multiplier U sing Verilog by Addanki Puma Ramesh, A. V. N. Tilak, A.M. Prasad.

Appendices

1. Verilog Code

```
module double_multiplier(
input_a,
input_b,
input_a_stb,
input_b_stb,
output_z_ack,
clk,
rst,
output_z,
output_z_stb,
input_a_ack,
input_b_ack);
input clk;
input rst;
input [63:0] input_a;
input input_a_stb;
output input_a_ack;
input [63:0] input_b;
input input_b_stb;
output input_b_ack;
output [63:0] output_z;
output output_z_stb;
input output_z_ack;
reg s_output_z_stb;
reg [63:0] s_output_z;
reg s_input_a_ack;
reg s_input_b_ack;
reg [3:0] state;
parameter get_a = 4'd0,
get_b = 4'd1,
unpack = 4'd2,
special_cases = 4'd3,
normalise_a = 4'd4,
normalise_b = 4'd5,
multiply_0 = 4'd6,
multiply_1 = 4'd7,
normalise_1 = 4'd8,
normalise_2 = 4'd9,
round = 4'd10,
pack = 4'd11,
put_z = 4'd12;
reg [63:0] a, b, z;
reg [52:0] a_m, b_m, z_m;
reg [12:0] a_e, b_e, z_e;
reg a_s, b_s, z_s;
reg guard, round_bit, sticky;
reg [107:0] product;
always @(posedge clk)
begin
case(state)
get_a:
begin
s_input_a_ack <= 1;
if (s_input_a_ack && input_a_stb) begin
a <= input_a;
s_input_a_ack <= 0;
state <= get_b;
end
end
get_b:
begin
s_input_b_ack <= 1;
if (s_input_b_ack && input_b_stb) begin
b <= input_b;
s_input_b_ack <= 0;
state <= unpack;
end
end
unpack:
begin
a_m <= a[51 : 0];
b_m <= b[51 : 0];
a_e <= a[62 : 52] - 1023;// true exponent
b_e <= b[62 : 52] - 1023;// true exponent
```

```
a_s <= a[63];
b_s <= b[63];
state <= special_cases;
end
special_cases:
begin
//if a is NaN or b is NaN return NaN ///
here we are all taking true exponents
if ((a_e == 1024 && a_m != 0) || (b_e ==
1024 && b_m != 0)) begin
z[63] <= 1;//quiet NAN if sign is +1//
z[62:52] <= 2047;// excluding zero//
z[51] <= 1;
z[50:0] <= 0;
state <= put_z;
//if a is inf return inf
end else if (a_e == 1024) begin
z[63] <= a_s ^ b_s;
z[62:52] <= 2047;
z[51:0] <= 0;
state <= put_z;
//if b is inf return inf
end else if (b_e == 1024) begin
z[63] <= a_s ^ b_s;
z[62:52] <= 2047;
z[51:0] <= 0;
//if a is zero return zero
end else if (($signed(a_e) == -1023) &&
(a_m == 0)) begin
z[63] <= a_s ^ b_s;
z[62:52] <= 0;
z[51:0] <= 0;
state <= put_z;
//if b is zero return zero
end else if (($signed(b_e) == -1023) &&
(b_m == 0)) begin
z[63] <= a_s ^ b_s;
z[62:52] <= 0;
z[51:0] <= 0;
state <= put_z;
end else begin
//Denormalised Number
if ($signed(a_e) == -1023) begin
a_e <= -1022;
end else begin
a_m[52] <= 1;
end
//Denormalised Number
if ($signed(b_e) == -1023) begin
b_e <= -1022;
end else begin
b_m[52] <= 1;
end
state <= normalise_a;
end
end
normalise_a:
begin
if (a_m[52]) begin
state <= normalise_b;
end else begin
a_m <= a_m << 1;
a_e <= a_e - 1;
end
end
normalise_b:
begin
if (b_m[52]) begin
state <= multiply_0;
end else begin
b_m <= b_m << 1;
b_e <= b_e - 1;
end
end
multiply_0:
begin
z_s <= a_s ^ b_s;
z_e <= a_e + b_e+1;
product <= a_m * b_m ;
state <= multiply_1;
```

```
end
multiply_1:
begin
z_m <= product[107:55];
guard <= product[54];
round_bit <= product[53];
sticky <= (product[52:0] != 0);
state <= normalise_1;
end
normalise_1:
begin
if (z_m[52] == 0) begin
z_e <= z_e - 1;
z_m <= z_m << 1;
z_m[0] <= guard;
guard <= round_bit;
round_bit <= 0;
end else begin
state <= pack;
end
end
pack:
begin
z[51 : 0] <= z_m[51:0];
z[62 : 52] <= z_e[11:0] + 1023;
z[63] <= z_s;
if ($signed(z_e) == -1022 && z_m[52]
== 0) begin
z[62 : 52] <= 0;
end
//if overflow occurs, return inf
if ($signed(z_e) > 1023) begin
z[51 : 0] <= 0;
z[62 : 52] <= 2047;
z[63] <= z_s;
end
state <= put_z;
end
put_z:
begin
s_output_z_stb <= 1;
s_output_z <= z;
if (s_output_z_stb && output_z_ack)
begin
s_output_z_stb <= 0;
state <= get_a;
end
end
endcase
if (rst == 1) begin
state <= get_a;
s_input_a_ack <= 0;
s_input_b_ack <= 0;
s_output_z_stb <= 0;
end
end
assign input_a_ack = s_input_a_ack;
assign input_b_ack = s_input_b_ack;
assign output_z_stb = s_output_z_stb;
assign output_z = s_output_z;
endmodule
```

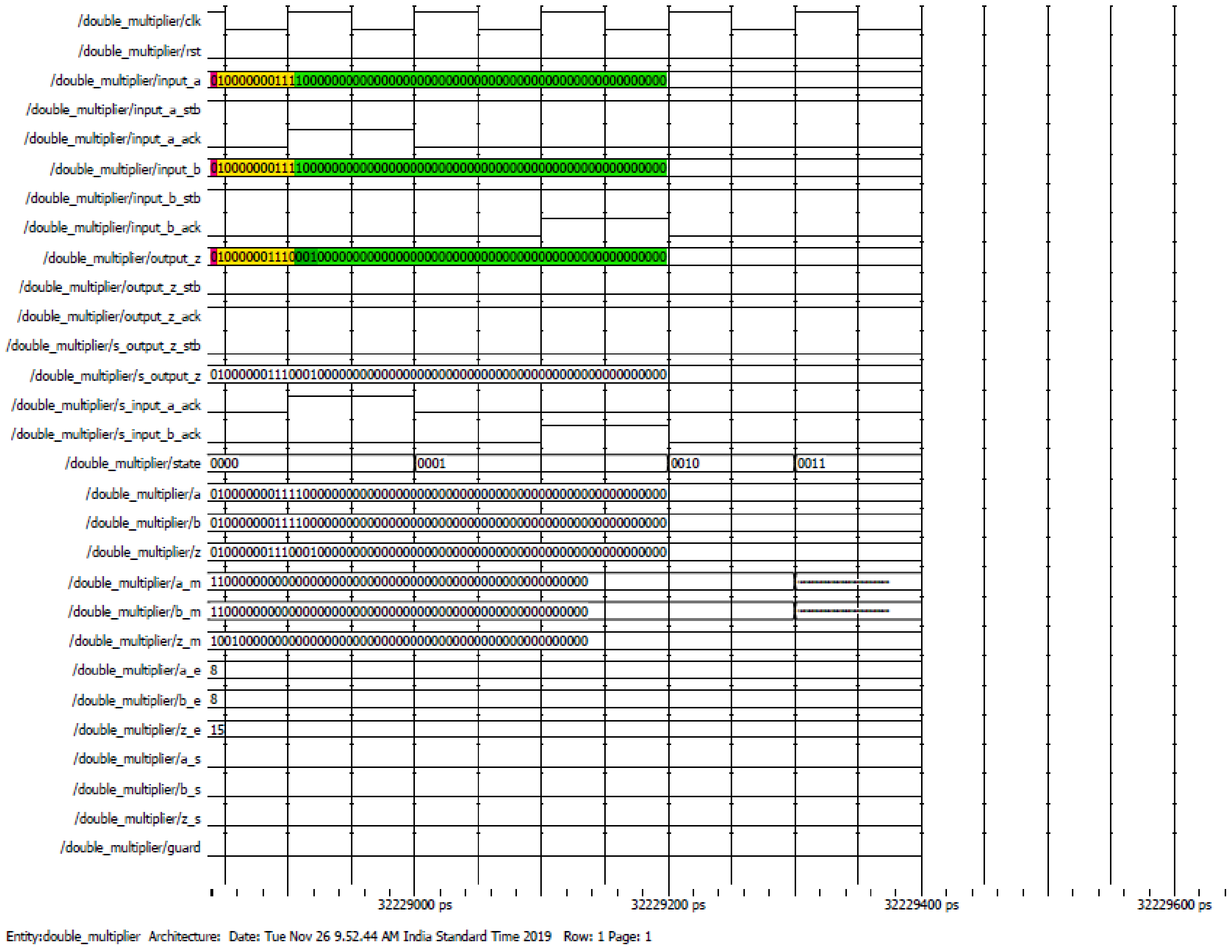
2. Simulation Outputs

Here, we have taken example as

Multiplicand is 1.5x10^{8} (1.1x2^{8})

Multiplier is 1.5x10^{8}(1.1x2^{8})

Product is 2.25x10^{16}(1.001x2^{15} or 10.01x2^{16})

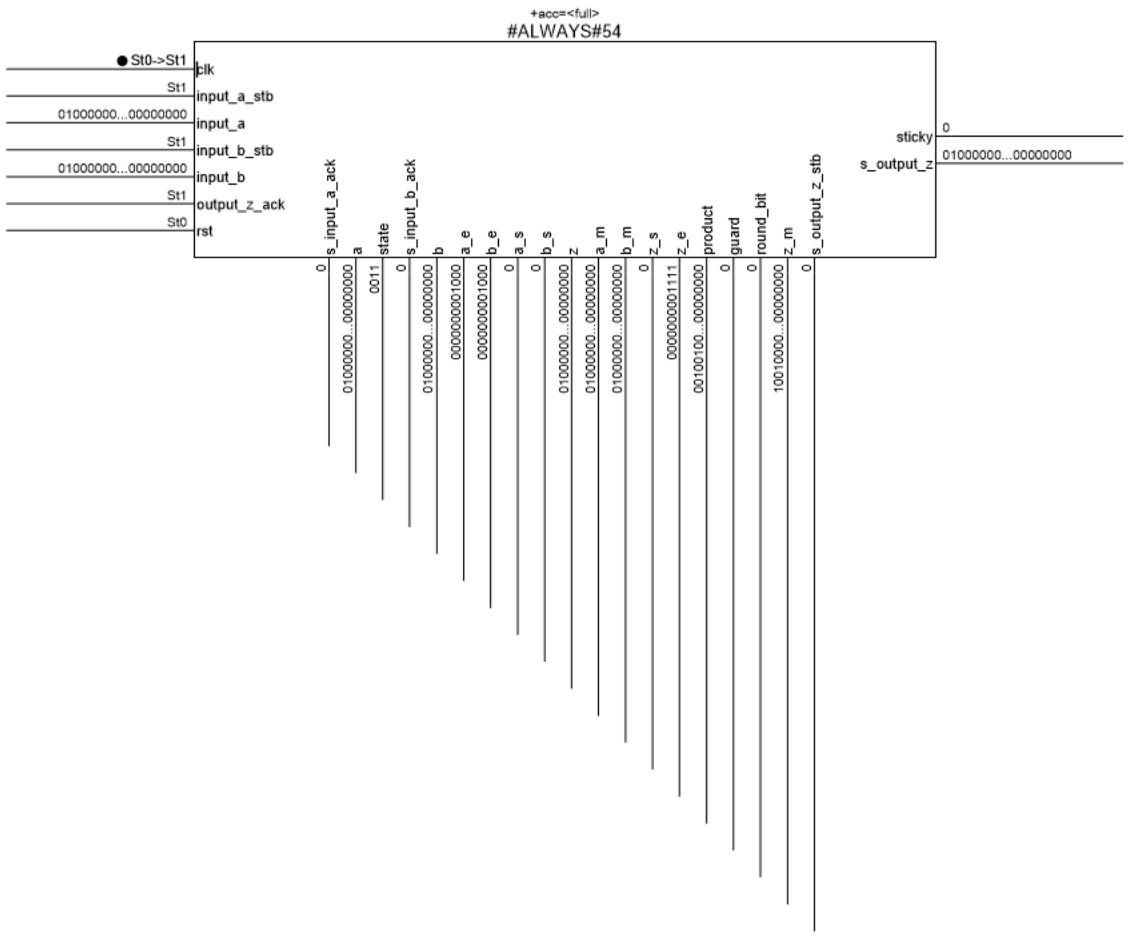


Entity:double_multiplier Architecture: Date: Tue Nov 26 9.52.44 AM India Standard Time 2019 Row: 1 Page: 1

3.Test bench:

```
`timescale 1ps / 1ps
module DFPM  ;

parameter multiply_1  = 4'b0111 ;
parameter get_a  = 4'b0000 ;
parameter get_b  = 4'b0001 ;
parameter put_z  = 4'b1010 ;
parameter normalise_1  = 4'b1000 ;
parameter normalise_a  = 4'b0100 ;
parameter unpack  = 4'b0010 ;
parameter pack  = 4'b1001 ;
parameter multiply_0  = 4'b0110 ;
parameter normalise_b  = 4'b0101 ;
parameter special_cases  = 4'b0011 ;
  reg    output_z_ack   ;
  reg    input_a_stb   ;
  reg  [63:0]  input_a   ;
  reg    rst   ;
  reg    input_b_stb   ;
  reg  [63:0]  input_b   ;
  wire    input_a_ack   ;
  wire    output_z_stb   ;
  wire  [63:0]  output_z   ;
  reg    clk   ;
  wire    input_b_ack   ;
  double_multiplier    #( multiply_1 , get_a , get_b ,
put_z , normalise_1 , normalise_a , unpack , pack ,
multiply_0 , normalise_b , special_cases  )
   DUT  (
       .output_z_ack (output_z_ack ) ,
      .input_a_stb (input_a_stb ) ,
      .input_a (input_a ) ,
      .rst (rst ) ,
      .input_b_stb (input_b_stb ) ,
      .input_b (input_b ) ,
      .input_a_ack (input_a_ack ) ,
      .output_z_stb (output_z_stb ) ,
      .output_z (output_z ) ,
      .clk (clk ) ,
      .input_b_ack (input_b_ack ) );



// "Clock Pattern" : dutyCycle = 50
// Start Time = 0 ps, End Time = 1 ns, Period = 100 ps
  initial
  begin
   repeat(10)
   begin
               clk  = 1'b1  ;
              #50  clk  = 1'b0  ;
              #50 ;
// 1 ns, repeat pattern in loop.
   end
  end


// "Constant Pattern"
// Start Time = 0 ps, End Time = 1 ns, Period = 0 ps
  initial
  begin
              rst  = 1'b0  ;
             # 1000 ;
// dumped values till 1 ns
  end


// "Constant Pattern"
// Start Time = 0 ps, End Time = 1 ns, Period = 0 ps
  initial
  begin
              input_a_stb  = 1'b1  ;
             # 1000 ;
// dumped values till 1 ns
  end


// "Constant Pattern"
// Start Time = 0 ps, End Time = 1 ns, Period = 0 ps
  initial
  begin
              input_b_stb  = 1'b1  ;
             # 1000 ;
// dumped values till 1 ns
  end


// "Constant Pattern"
// Start Time = 0 ps, End Time = 1 ns, Period = 0 ps
  initial
  begin
              if (output_z_stb  != (1'b0  )) $display($time, "
test case failed");
             # 1000 ;
// dumped values till 1 ns
  end

  initial
            #2000 $stop;
endmodule
```

4.Implementation Block Diagram



****