|  |
| --- |
| 32-bit FLOATING POINT MULTIPLIER IMPLEMENTATION |

*Version 1.0 (Wednesday, April 17, 2019)*

Preface:

We[[1]](#footnote-0) have created this multiplier to handle the multiplication process of two 32-bit IEEE 754 based numbers as a part of FPU project[[2]](#footnote-1).

Content:

1. IEEE 754 representation ....................................................................1
2. The implementation JOURNY ...............................................................2
   1. calculating fraction .................................................................2
   2. Calculating exponent ................................................................
   3. THE REASEONS WHY WE WERE SO LUCKY IMPLEMETING THIS .......
3. Special cases .......................................................................................
4. VERILOG CODE ........................................................................................

APPENDEX A. Sharing the happiness of a working code (results) ..

APPENDEX B. OUR TOOLKIT

APPENDEX C. REFERENCES

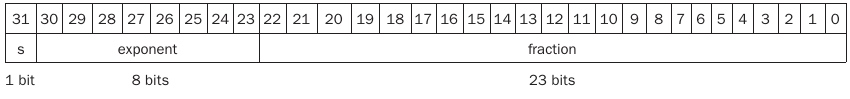
chapter 1:

“IEEE 754” representation

“This kind of refreshing part for the sake of comprehension.”

Floating-Point Representation:

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.



In general, floating-point numbers are of the form

DeepinScreenshot_select-area_20190429043519

These formats go beyond MIPS. They are part of the IEEE 754 floating-point standard, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

Before we go on board we have to take these notes with us:

1. Number must be normalized in an understandable language ( 1.01000 \* 2^E not 101.000\* 2^E-3 )
2. Exponents are biased by 127 which means 0 = 127 to get the the real exponent form IEEE754 one subtract 127 from it.

chapter 2:

Our implementation journey[[3]](#footnote-2)

“Our implementation wasn’t the most complicated ever but we tend to call it a “journey” because we had explored a new world”

2.1 Calculating fraction:

To calculate fractions we first have to notice that the input has the form of “1.(mantissa)\*2 exponent\*(-1) sign “ in this section we only concerned with the “1.(mantissa)” part we can express it in other form as “1.0 + 0.mantissa->(also knows as fraction we may use these terms interchangeably so don’t get confused)” multiplying each input we can easily notice that the result has the form

|  |
| --- |
| **Formula 1:**  Multiplying each input gives  Result = (1+ fraction1 + fraction2 + fraction1 \* fraction2)  Resulting fraction = Result - 1 or  **( fraction1 + fraction2 + fraction1 \* fraction2 )** |

With the use of formula 1 we can calculate our resulting fraction in a single Verilog line more on this later at (Chapter 4 & Appendix .A)

2.2 Calculating Exponent:

To calculate exponents we just add them together as we learned at the primary school nothing amazing here.

|  |
| --- |
| **Formula 2:**  Resulting exponent = (exponent1 + exponent2) |

2.3 THE REASEONS WHY WE WERE SO LUCKY IMPLEMETING THIS:

## “WELCOME TO THE REALM OF LUCKY PEOPLE”, THE DUKE SAID.

We as well as anyone who was implementing the multiplication process in FPU project probably were the people with a bunch of luck, here is a list of some reasons.

1. We don’t have to build a complicated algorithm with many cases to handle normalization process; we can just check for the last if it’s high shift the fraction to the right one step then increase the exponent by one if it’s not just do nothing.
2. IEEE 754 has an out of the box biased exponents system which means that we just avoided the overhead of subtraction and adding exponents will just do a decent job
3. Multiplication process as FSM is synthesizable so using multiplication is completely acceptable.

chapter 3:

SPECIAL CASES

“love them , hate them, we can’t live without handling them”

chapter 4:

Verilog code

“Coding stuff...”

|  |
| --- |
| module multi(  output logic [31:0] op, //O/P  input logic [31:0] a,  input logic [31:0] b  );  logic [22:0]Ofraction\_HI; //fraction will neglict any bit after ofraction[22]  logic [22:0]Ofraction\_LO;  logic Osign;  logic [7:0]Opower;  logic [22:0]Afraction ;  logic [22:0]Bfraction ;  logic [7:0] Apower;  logic [7:0] Bpower;  logic Asign;  logic Bsign;  logic [1:0]carryExp;  logic [2:0]carryFra;  always @(a or b) begin  {Asign,Apower,Afraction} = a;  {Bsign,Bpower,Bfraction} = b;    // adding brances to solve an issue with order of adding  {carryFra,Ofraction\_HI,Ofraction\_LO} = ((Afraction\*Bfraction) + (Afraction<<23) + (Bfraction<<23));    // exponent is biased by 127 so (000000011)  // represented in (100000010)  {carryExp,Opower} = Apower + Bpower - 127 + carryFra;  // TODO move carry fraction to if condition as adding one  // to handle when f1\*f2+f1+f2>2  Osign = Asign ^ Bsign;  // normlizing number for case of ofraction > 1  if (carryFra)  Ofraction\_HI = Ofraction\_HI>>1;  //check the exponent  if (carryExp)  $display("overflow");  op = {Osign , Opower , Ofraction\_HI};  end    endmodule |

1. **Team members:**

   - Hussien Mostafa

   - Abdulrahman Ragab

   - Abdullah Khaled

   - Abdallah Mohamed

   - Mahmoud Hassan

   **Supervisor:**

   - Dr. Muhammad Mahmoud Muhammad Ibrahim [↑](#footnote-ref-0)
2. **FPU project:** the floating point unit to be used in our implementation of MIPS based processor [↑](#footnote-ref-1)
3. side-note: we roughly assume that the reader have the basic knowledge of Algebra and programming basics [↑](#footnote-ref-2)