Pipelined Double Precision Floating Point Multiplier

Patha Yedukondalu cs23m005

Samyuktha M cs23s023

Table of Contents

Pipelined Double Precision Floating Point Multiplier 1

- **1. Introduction** 3
- **2. Preliminaries** 3
  - **2.1. IEEE 754 Standard for Representation of Floating Points** 3
  - **2.2. IEEE 754 Floating Point Multiplication** 4
- **3. Literature Survey** 5
- **4. Our Approach** 5
- **5. Tools Used** 7
  - **5.1. Bluespec Development Workstation** 7
  - **5.2. Yosys** 7
- **6.Testing** 8
- **7. Synthesis** 10
- **8. Results** 10
  - **8.1 Area Calculation** 10
- **9. Conclusion** 11
- **10. References** 11

1. Introduction

In the realm of computer architecture and numerical computation, precision plays a pivotal role in ensuring the accuracy of mathematical operations. One fundamental component that greatly influences precision is the floating-point multiplier, a crucial element in processors designed to handle real numbers with a wide range of magnitudes. The need for double precision floating-point multipliers arises from the demand for increased accuracy in scientific, engineering, and financial applications, where computations involve extremely large or small numbers and require meticulous attention to detail.

Double precision, referring to the use of 64 bits to represent a floating-point number, provides a higher level of precision compared to single precision (32 bits). In scenarios where the difference between values is minute or where extensive calculations are involved, the enhanced precision of double precision floating-point multipliers becomes indispensable. This is particularly evident in scientific simulations, climate modelling, financial modelling, and other fields where the consequences of rounding errors or inaccuracies can have profound implications.

Furthermore, the adoption of double precision floating-point arithmetic aligns with the increasing demand for computational power in contemporary applications. As processors strive to handle complex tasks with ever-growing data sets, the ability to accurately represent and manipulate real numbers becomes essential. Double precision arithmetic ensures that numerical results maintain a level of precision that is not only crucial for scientific integrity but also for the reliability of critical calculations in various domains

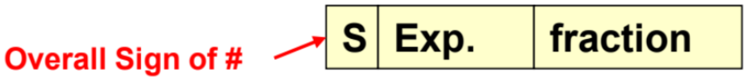
2. Preliminaries

2.1. IEEE 754 Standard for Representation of Floating Points

IEEE 754 outlines four representation formats: single precision (32-bit), double precision (64-bit), single extended (≥ 43 bits), and double extended precisions (≥ 79 bits). In accordance with this standard, floating-point numbers consist of three components: a sign, an exponent, and a mantissa. The mantissa incorporates an implied leading bit, and the remaining bits are fractional. The split up of bits for each of the component vary according to the representation formats.

IEEE 754 standard establishes a system for representing floating-point numbers, including special values like infinity and NaN, as well as defining a set of operations, rounding modes, and handling specific cases.

Double Precision Floating Point Representation		
Component	**# of bits**	**Representation**
Sign	1	
Exponent	11	Excess-N Notation
Mantissa/Significand	52	



Floating Point representation uses the following format,

$$\pm b.bbbb * 2^{\pm exp}$$

Floating point numbers are normalised for storage. The leading bit is 1 for normalised representation and is actually not stored but assumed. Thus normalized FP format is:

$$\pm 1.bbbbbb * 2^{\pm exp}$$

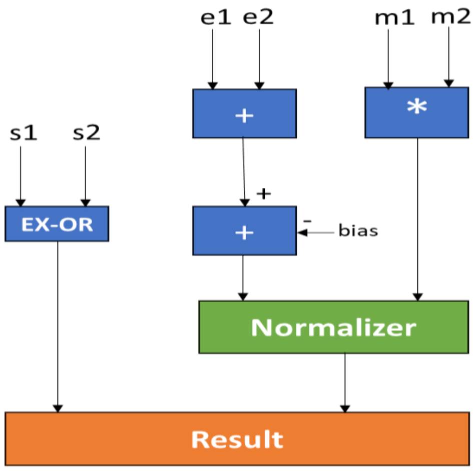
The general form is given by:

$$\mathbf{(-1)^{sign} \times 1.M \times 2^{exponent-1023}}$$

where M is the binary representation of the significand.

2.2. IEEE 754 Floating Point Multiplication

Let s1, e1, m1 correspond to the sign, exponent and mantissa of the multiplicand respectively and s2, e2, m2 correspond to the sign, exponent and mantissa of the multiplier respectively.



Sign of the product:

Sign of the product is simply the ex-or of the sign bits of the multiplier and the multiplicand.

Exponent of the product:

Exponent of the product is the addition of the exponents of the multiplier and the multiplicand and the bias is subtracted with the added exponent.

Mantissa of the product:

Mantissa of the product involve binary multiplication computing the partial products of the multiplier and the multiplicand. This has been implemented using the carry save multiplier.

Normalisation and Rounding:

The Result is normalised to the specified format and rounded using a rounding technique

3. Literature Survey

Paschalakis et al [1] presented a low-cost implementation of the FPGA floating-point arithmetic circuits for all the common operations, i.e. addition/subtraction, multiplication, division and square root. Their implementation is not pipelined and with a latency of ten cycles. The multiplier unit defined in this paper also included an exception handling mechanism and rounding techniques to attain reasonable precision and accuracy. Manolopoulos et al in [2] implemented a multi-mode floating point multiplier for all precision formats specified by IEEE 754-2008. Their parallel double precision multiplier had a latency of only two cycles. An example implementation on VLSi to verify the design was also provided in the paper. In [2] the authors adopt the divide-and-conquer technique for carrying out the double precision multiplication. The Mantissa is extended to 58-bits by padding additional zeros and divided into two parts of 27-bits and then the multiplication is applied. The partial product is calculated in the first cycle and in the second cycle the design performs addition, rounding and normalization.

Banescu et al in [3] studied the automated generation of multipliers using the embedded multipliers and adders present in the DSP blocks of current FPGAs. The optimization of such multipliers is expressed as a tiling problem, where a tile represents a hardware multiplier, and super-tiles represent combinations of several hardware multipliers and adders, making efficient use of the DSP internal resources. Saini et al in [4] introduced a pipeline floating point arithmetic logic unit (ALU). Pipeline has been used to give high performance and throughput to arithmetic operation. They also conducted a comparative analysis between pipeline and sequential approach in terms of speed, power, throughput, latency and chip area.

Other notable works involve using Wallace trees and its derivatives which were outlined in [5], [6],[7], [8]. The most common approach was the booth's multiplier and was adopted by authors in [9], [10], [11], [12]. Our approach involves using a carry save multiplier to compute the partial products through a pipelined architecture.

4. Our Approach

Our approach uses a five-stage pipeline in order to achieve high throughput. We have employed a divide and conquer strategy where the partial products on a split up of 6-bits are computed to reach the final result.

The following flags are used to notify the user and to handle the exceptions

1. Overflow Flag: This flag is set when the when the result of a multiplication operation exceeds the maximum representable range. This flag can only be set after multiplying the multiplier and the multiplicand.

2. Underflow Flag: This flag is set when the when the result of a multiplication operation lesser than the minimum representable range. This flag can only be set after multiplying the multiplier and the multiplicand.

3. Nan Flag: This flag is set when either of the multiplier or a multiplicand is NaN. This flag can be set prior to multiplication of the multiplier and the multiplicand.
4. Zero Flag: This flag is set when either of the multiplier or a multiplicand is Zero. This flag can be set prior to multiplication of the multiplier and the multiplicand.
5. Infinity Flag: This flag is set when either of the multiplier or a multiplicand is plus or minus infinity. This flag can be set prior to multiplication of the multiplier and the multiplicand.
6. Sub-Normal Flag: This flag is set when either of the multiplier or a multiplicand is a sub-normal number. This flag can be set prior to multiplication of the multiplier and the multiplicand.

In the Stage-I of the pipeline, the multiplier and the multiplicand are unpacked and are provided as input. The sign computation and the exponent computation of the product are carried out and few of the flags that can be set prior to the multiplication are set. The 52-bit mantissa are split into 6-bits each forming nearly 8 and a half partial products. 2 zeroes are padded to get 54-bits thereby forming 9 partial products. 9 partial products of the mantissas of the multiplier and the multiplicand, replicating the hardware 9 times are computed and passed as output to the next stage.

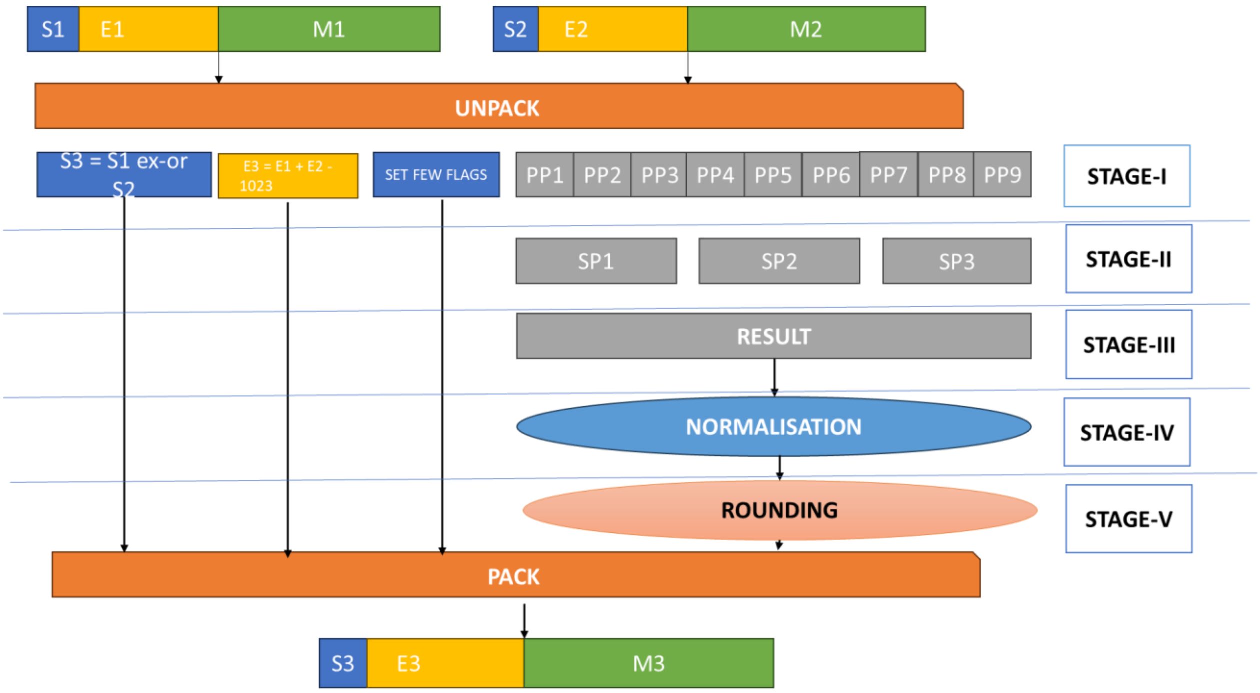
In the Stage-II of the pipeline, the results of the 9 partial products are added three at a time, requiring to repeat the hardware 3 time resulting in three sums of the 9 ppartial products. These three sums along with the sign and exponents computed in Stage-I are passed on to the next stage of the pipeline.

In the Stage-III of the pipeline, the three sums obtained in the previous stage are added to obtain the final result in 105 bits. The final sum along with the sign and exponents computed in Stage-I are passed on to the next stage of the pipeline.

In the Stage-IV of the pipeline, the result is normalised and the mantissa and the exponent components of the product of the multiplier and the multiplicand are adjusted. The Normalised result along with the sign and exponents computed in Stage-I are passed on to the next stage of the pipeline.

In the Stage-V of the pipeline, the product which is in 105 bits is rounded by chopping of the 53 least significant bits to obtain the final result. The final result along with the sign and exponents computed in Stage-I are packed to be displayed as the output.

Our design specification included importing the FloatingPoint package defined as part of the bluespec. The details of the pipeline are also outlined in the figure below.



5. Tools Used

5.1. Bluespec Development Workstation

The BSC Development Workstation (BDW) is a comprehensive graphical environment that enables users to perform a range of tasks, including creating, editing, compiling, simulating, analyzing, and debugging BSV/BH designs using BSC. BDW has the capability to interface with waveform viewers like GtkWave, facilitating source-level debugging of simulations. This includes the ability to:

- Visualize signal values in terms of source-level types, such as enums, structs, and tagged unions, rather than single bit vectors.
- Incorporate sets of signals into an attached waveform viewer for a chosen rule, method, or module instance in the design hierarchy.
- Examine CAN_FIRE and WILL_FIRE for rules.
- Inspect RDY, EN, argument, and result ports for a method.

Moreover, BDW can establish connections with text editors like Emacs or Vim, enabling users to quickly access the source code for various elements such as types, functions, rules, modules, instantiations, and more directly from the diverse viewer windows within BDW.

5.2. Yosys

Yosys is an open-source framework for Verilog RTL (Register Transfer Level) synthesis. RTL synthesis is the process of converting high-level hardware description language (HDL) code, such as Verilog or VHDL, into a netlist representation that can be used for further stages of the digital design flow, including place and route for FPGA or ASIC implementation. Some key benefits that Yosys offers are:

- **Open Source**: Yosys is an open-source tool

- **RTL Synthesis**: The primary purpose of Yosys is RTL synthesis. It takes Verilog code as input and generates a netlist, which is a low-level representation of the design in terms of logical gates and flip-flops.
- **Support for Various FPGA Architectures**: Yosys can target different FPGA architectures, allowing designers to implement their designs on various FPGA devices.
- **Scripting Interface**: Yosys provides a scripting interface that allows users to automate the synthesis process. This is particularly useful for complex designs or when running multiple iterations of synthesis with different settings.
- **Formal Verification**: Yosys includes some formal verification capabilities, which can be used to check certain properties of the design and ensure correctness

6.Testing

The floating-point multiplier implementation was tested against the following testcases to ensure right handling of

- Standard/Normal arithmetic
- Subnormal arithmetic
- Infinite arithmetic
- NaN arithmetic
- Zero arithmetic

The test cases were generated using the C language and the test cases used include

1. Basic Multiplication:
    - Multiplying positive numbers: Test the multiplication of two positive double-precision floating-point numbers.
    - Multiplying negative numbers: Test the multiplication of two negative double-precision floating-point numbers.
    - Multiply positive and negative numbers: Test the multiplication of a positive and a negative double-precision floating-point number
    - Example: Input: (1.5, 2.0) Expected Output: 3.5
2. Zero Multiplication:
    - Multiply any double-precision floating-point number by zero and vice versa.
    - Example: - Input: (0.0, 3.14159) Expected Output: 0.0
3. Overflow Test:
    - Multiply two very large positive numbers and check if overflow is handled correctly.
    - Example: Input: (1.0e308, 2.0e308) (maximum representable double-precision numbers) Expected Output: Infinity (or an appropriate overflow representation)
4. Underflow Test:
    - Multiply two very small positive numbers and check if underflow is handled correctly.
    - Example: Input: (1.0e-308, 2.0e-308) (very small double-precision numbers) Expected Output: A denormalized number or 0.0 (depending on the handling of underflow)

5. Mixed Precision Test:
    - Values of different precisions are multiplied
    - Example - Input: (1.23456789, 0.00000001) Expected Output: A result close to 0.0000000123456789
6. Large Magnitude Multiplication:
    - Multiplication that results in a larger number but not causing overflow
    - Example - Input: (1.0e200, 1.0e200) - Expected Output: A very large result

7. Test for Denormalized Numbers:
    - Multiply denormalized numbers: Test the multiplication of two denormalized (subnormal) double-precision floating-point numbers.
    - Example: - Input: (1.0e-323, 2.0e-323) (smallest representable positive double-precision numbers) Expected Output: A denormalized number or 4.9406564584124654e-323
8. Test for NaN Handling:
    - Test the multiplication of a double-precision floating-point number by NaN (Not a Number).
    - Example: Input: (NaN, 3.14) or (2.0, NaN) Expected Output: NaN (Not-a-Number)
9. Infinity Handling:
    - Multiply a finite number by positive or negative infinity.
    - Example : Input: (Infinity, 5.0) or (4.0, -Infinity) Expected Output: Infinity or -Infinity
10. Rounding and Precision:
    - Test for round-off errors by multiplying numbers that are very close together but not exactly representable
    - Example: Input: (0.1, 0.2) Expected Output: 0.02 (Double-precision rounding behaviour)
11. Special Cases:
    - Test with edge cases, such as positive and negative zeros, subnormal numbers, and special values like +/- 0, +/- Infinity, etc.
12. Random Test Cases:
    - Generate random double-precision numbers and verify the correctness of the multiplication result.

Test Case	Status
Basic Multiplication	Pass
Zero Multiplication	Pass
Overflow Test	Pass
Underflow Test	Pass
Mixed Precision Test	Pass
Large Magnitude Multiplication	Pass
Test for Denormalized Numbers	Pass
Test for NaN Handling	Pass
Infinity Handling	Pass

Rounding and Precision	Pass
Special Cases	Pass
Random Test Cases	Pass

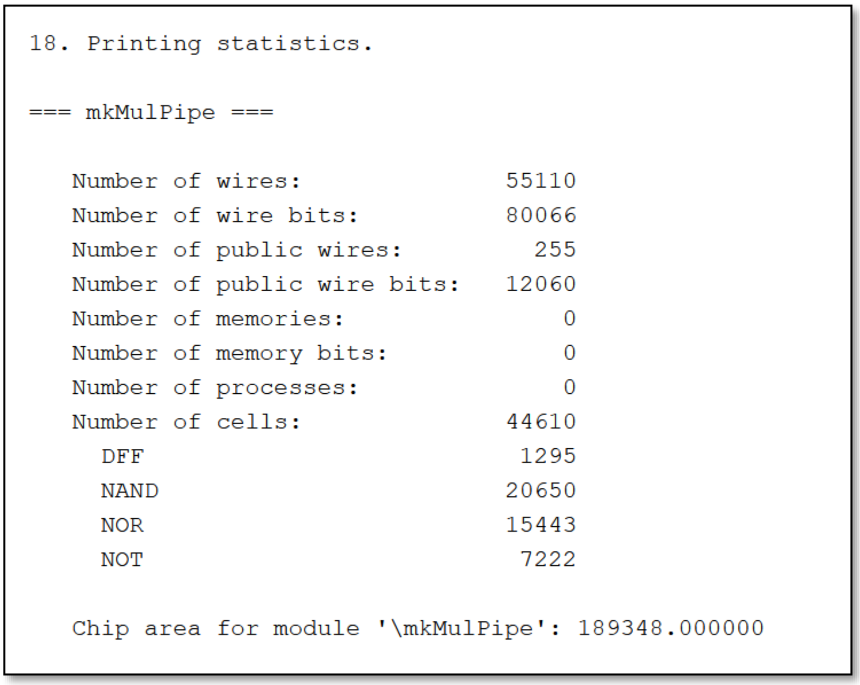
7. Synthesis

The implemented bluespec code is synthsized using the Yosys tool to obtain the netlist. The netlist is also uploaded on the github link under the results folder in the name ‘netlist.v’ and ‘abcd.dot’.

8. Results

8.1 Area Calculation

The area is computed using the Yosys tool and the result is displayed in the figure below:

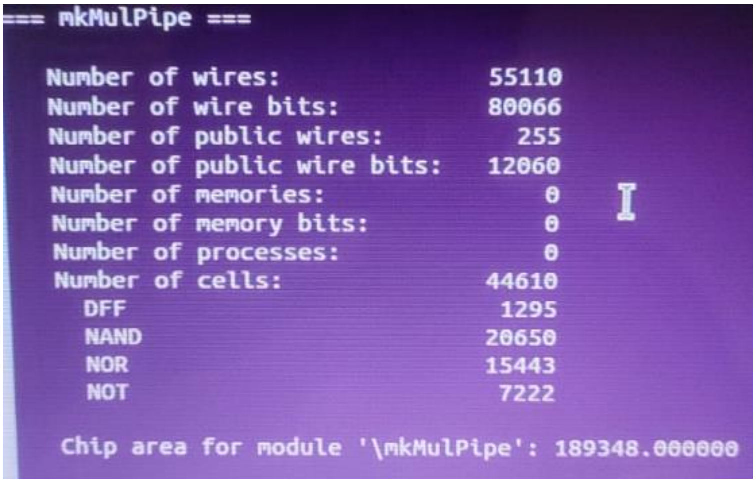


```
18. Printing statistics.

=== mkMulPipe ===

   Number of wires:              55110
   Number of wire bits:          80066
   Number of public wires:         255
   Number of public wire bits:   12060
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              44610
     DFF                          1295
     NAND                        20650
     NOR                         15443
     NOT                          7222

   Chip area for module '\mkMulPipe': 189348.000000
```



```
=== mkMulPipe ===

   Number of wires:              55110
   Number of wire bits:          80066
   Number of public wires:         255
   Number of public wire bits:   12060
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              44610
     DFF                          1295
     NAND                        20650
     NOR                         15443
     NOT                          7222

   Chip area for module '\mkMulPipe': 189348.000000
```

9. Conclusion

In conclusion, the development and synthesis of the pipelined double-precision floating-point multiplier represent a significant learning in understanding the ease of use of bluespec in VLSI. Through testing and synthesis, we have not only validated the functionality of the multiplier but also obtained insights into its performance characteristics.

The calculated area metrics provide a tangible measure of the hardware resources required for the implementation, offering information for assessing the design's efficiency and scalability. This synthesis process, coupled with comprehensive testing, ensures that the pipelined double-precision floating-point multiplier meets the specified requirements and can be seamlessly integrated into larger digital systems.

10. References

1. S. Paschalakis and P. Lee, "Double precision floating-point arithmetic on FPGAs," Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798), Tokyo, Japan, 2003, pp. 352-358, doi: 10.1109/FPT.2003.1275775.

2. K. Manolopoulos, D. Reisis and V. A. Chouliaras, "An efficient multiple precision floating-point multiplier," 2011 18th IEEE International Conference on Electronics, Circuits, and Systems, Beirut, Lebanon, 2011, pp. 153-156, doi: 10.1109/ICECS.2011.6122237.

3. Banescu, S., De Dinechin, F., Pasca, B. and Tudoran, R., 2011. Multipliers for floating-point double precision and beyond on FPGAs. *ACM SIGARCH Computer Architecture News*, *38*(4), pp.73-79.

4. Saini, R. and Daruwala, R.D., 2013. Efficient Implementation of Pipelined Double Precision Floating Point Multiplier. *International Journal of Engineering Research and Applications*, *3*(1), pp.1676-1679.

5. Jain, A., Jain, R. and Jain, J., 2018, December. Design of reversible single precision and double precision floating point multipliers. In 2018 International Conference on Advanced Computation and Telecommunication (ICACAT) (pp. 1-4). IEEE.

6. Jimenez Perez, A., 2021. Design and Comparative Analysis of Several Types of Fixed and Simple Precision Floating Point Multipliers. Instituto de Ingeniería y Tecnología.

7. Liu, Z., Ma, S. and Guo, Y., 2014. An efficient floating-point multiplier for digital signal processors. IEICE Electronics Express, 11(6), pp.20140078-20140078.

8. Luo, Z. and Martonosi, M., 2000. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. IEEE Transactions on Computers, 49(3), pp.208-218.

9. Jaiswal, M.K. and So, H.K.H., 2015, October. Dual-mode double precision/two-parallel single precision floating point multiplier architecture. In 2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC) (pp. 213-218). IEEE.

10. Ramteke, Pallavi, N. N. Mhala, and P. R. Lakhe. "An efficient implementation of double precision floating point multiplier using booth algorithm." Int J Adv Res Electr Electron Instrum Eng 3, no. 7 (2014).

11. Paruthi, Puneet, Tanvi Kumar, and Himanshu Singh. "Simulation of IEEE 754 standard double precision multiplier using booth techniques." International Journal of Engineering Research and Applications (IJERA) 2 (2012): 1761-1766.

12. Koppala, N. and RohitSreerama, P., 2012. Performance comparison of fast multipliers implemented on variable precision floating point multiplication algorithm. International Journal of Computer Applications in Engineering Sciences, 2(2).