RESEARCH ARTICLE OPEN ACCESS

Implementation of The Ieee Standard Binary Floating-Point Arithmetic Unit

Dr. R. Prakash Rao
Associate Professor, Electronics and Communication Engineering,
Matrusri Engineering College, #16-1-486, Saidabad, Hyderabad-500059, India.

Abstract:

A floating-point system can be used to represent, with a fixed number of digits, numbers of different orders of magnitude: e.g. the distance between galaxies can be expressed with the same unit of length. The result of this dynamic range is that the numbers that can be represented are not uniformly spaced; the difference between two consecutive representable numbers grows with the chosen scale. Over the years, a variety of floating-point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard. The speed of floating-point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system. Hence, in this work the IEEE standard binary floating-point arithmetic is implemented.

KEYWORDS: **Fixed Number of Digits, Ieee 754 Standard, Flops, Floating-Point System.**

I. INTRODUCTION

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely- used standard for floating-point computation and is followed by many CPU and FPU implementations. The result of multiplying two FP numbers can be described as multiplying their significands and adding their exponents. The resultant sign S is S1+S2, the resultant significand p is the adjusted product of p1.p2 and the resultant exponent E is the adjusted E1+E2+bias. In order to perform floating-point multiplication, a simple algorithm is realized as-

- Add the exponents and subtract 127.
- Multiply the mantissas and determine
- the sign of the result. Normalize the
- resulting value, if necessary.

II. ALGORITHMS AND DESIGN OF FLOATING POINT MULTIPLICATION

Figure.1 shows how the floating point multiplication is performed. Floating-point multiplication is inherently easier to design than floating-point addition or subtraction. Multiplication requires integer addition of operand exponents and integer multiplication of significands that facilitate normalization when multiplying normalized significands.

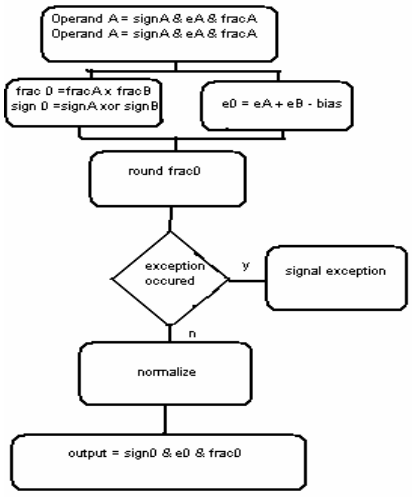

Figure.1: Basic design of Floating Point Multiplier

These independent operations within a multiplier make it ideal for pipelining. In

floating point multiplication the following three steps can be done:

- Unpack the operands, re-insert the hidden bit, and check for any exceptions on the operands (such as zeros or NaNs).
- Multiplication of the significands, calculation of the sign of the two significands, and addition of the exponents take place.
- The final result needs to be normalized and the exponent adjusted before packing and removing the hidden bit.

Multiplication does not require shifting of the significands or adjustment of the exponents as in the adder unit until the final stage for normalization purposes. For the basic summation of partial products in a floating-point multiplication represented in scientific notation (significant multiplied by the radix to some power), one multiplies the two significands and adds the two radix powers. Normalization of the significant ensures the decimal point of the significant has an exactly one significant digit to the left of it which may or may not need to be done [1].

III. 24-BIT PIPELINED INTEGER MULTIPLIER

The floating-point multiplier block diagram can be seen in Figure 2.

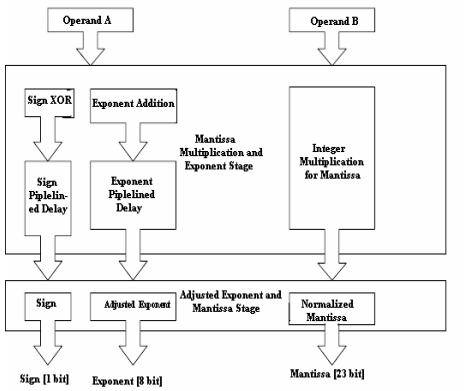

Figure 2: Pipelined Multiplier Block Diagram

The pipelined floating-point multiplier generates a product every clock when the pipeline has completely filled, and has a latency of 13 cycles. The pipeline stages for the multiplier are much simpler in comparison to the adder stages. Twelve of the 13 stages are used for the computation of the integer multiply. By simply relying upon VHDL, synthesis tools for the creation of the multiplication produced a design which was deemed unacceptable considering the planned resource budget. An alternative integer multiplier was created using a parameterized multiplier generation program. The generated 24X24 integer multiplier utilizes Booth recoding and pipeline stages to preserve routing, timing, and size of the multiplier. Two bits of the multiplier are issued at a time for twelve consecutive clock cycles, starting with the lowest two bits.

Figure 2 illustrates the pipeline multiplier stages for the floating-point multiplier. The exponent and mantissa operations can be performed concurrently until the final stage where normalization takes place. In floating-point multiplication, the exponents must be added together as they are in this implementation during the first stages. The result from the exponent addition continues through a pipeline delay until the mantissa result completes. Carry-out logic from the mantissa multiplication informs the control logic not to perform a 1-bit shift since the implied one exists. Note that the exponent must continue through several pipeline delays that require registered logic [2].

IV. IMPLEMENTATION IN VHDL

The multiplier VHDL consists of several different components that rely on a clocked process and registered signals. The components consist of a pipeline delay element, a 9- bit adder and a 24X24 pipelined integer multiplier. The VHDL clocked process provides much of the glue logic for the components used in order to ensure signals to each component are registered properly and to avoid timing hazards. The pipelined integer multiplier, for instance, requires that the inputs be registered for expected results. The inputs to the floating-point multiplier need to be checked for a possible zero outcome and assert a flag through the pipeline to indicate a zero value be given as the result during the last stage in the pipeline. The VHDL code provides concurrent operations for some of the initial stages. As the mantissa undergoes integer multiplication, calculations on

the exponent are done and passed through a pipeline delay to remain synchronized with the integer multiplier data. The VHDL used in the multiplier differs from the 32-bit floating- point pipelined adder in that no state machines are required for the multiplier. Instead, the VHDL provides minimal control logic to ensure the components are given data on the correct cycles.

4.1 Mantissa Multiplication and Exponent Addition

The multiplier undergoes two separate, parallel operations during the first 12 stages. One of the operations includes multiplying the two 24-bit mantissa values using the 24X24 pipelined integer multiplier. The multiplier generates the result on the thirteenth clock cycle. During the mantissa calculation, the exponent addition takes place using the 9-bit integer adder component. Nine bits, instead of eight, are used to handle carry-out situations. The carry-out bit provides important information used in the final stage of the floating-point multiplier to handle exponent biasing adjustments. Since the exponent calculation does not require more than a clock cycle, a pipeline delay component delays the calculated exponent result until the last stage when the bias adjustments are ready to be done. In addition, two smaller logic operations take place. The first determines if either of the input operands are zero. If so, a special zero-flag needs to be set. The second uses XOR logic to determine the resulting sign bit of the two input operands [3]. The zero-flag and sign bit need to be delayed as well until the last stage in the floating-point multiplier. All data going through the pipelined delay must continue to be synchronized with operand B going through the pipelined integer multiplier.

4.2 Exponent Adjustment and Product Assembly Stage

The last stage receives the data from the pipelined integer multiplier and the other pipeline delay elements. The stage logic checks the zero-flag bit to see if the output is simply a zero. Otherwise, a one in the most significant bit of the mantissa indicates the resulting mantissa value has already been normalized. If not, one and the mantissa output shifted by one must adjust the exponent. The exponent undergoes subtraction to remove an extra biasing factor from the addition of an earlier stage in the pipeline. Depending on the most significant bit of the mantissa, different values are subtracted from the exponent. The final stage assigns the resulting values to the output signals of the floating-point multiplier.

4.3 Stages in Single Precision Floating Point Multiplier

The standard floating point multiplier has several stages:

- Prenormalizing,
- Multiplying,
- Postnormalizing,
- Shifting
- Rounding

All of the stages involve multiple steps, but some stages are more complex than others. Each is described in its own section [3].

Prenormalizing

Recall that, in IEEE 754 format, normalized numbers have an implicit leading 1, and that denormalized numbers do not. Additionally, recall that denormalized numbers use leading zeros to increase the range of the exponent. To keep the multiplication stage simple, both inputs are converted into the same form.

Multiplying

The multiplication stage involves three parts: multiplying *mc* and *md*, adding *ec* and *ed*, and detecting tininess. Multiplying the mantissas ñ The two mantissas are already in a standard form due to the prenormalization stage.

Postnormalizing

In the postnormalization stage, the multiplier normalizes the product and returns MN

Shifting

If the product is a denormalized number, then it might need to be shifted to the right to provide the appropriate number of leading zeros to indicate the correct exponent.

Rounding

The product of two n-bit numbers has the potential of being 2(n+1) bits wide. The result of floating point multiplication, however, must fit into the same n bits as the multiplier and the multiplicand. This, of course, often leads to loss of precision. The IEEE standard attempted to keep this loss as minimal as possible with the introduction of standard rounding modes. [16]

When all are enabled, the multiplier supports all IEEE rounding modes: round to nearest even, round to zero, round to positive infinity, and round to negative infinity [4].

3.2.1Special Case Path

The multiplier cannot always determine a result by simply doing a multiplication. There are certain inputs that require the multiplier to take special action. The multiplier performs this action in parallel with the regular multiplication, and chooses this special result in cases in which it is required.

Not a number (NaN) - The IEEE 754 Standard specifies that an implementation will return a NaN that is given to it as input, or either one if both inputs are NaN's. The multiplier can be configured to return either the first NaN or the higher of the two. The Intel Pentium series returns the higher of the two NaNís and, as this multiplier was tested using a processor from that series, the multiplier is by default set to do the same.

Infinity - Nearly anything multiplied by infinity is properly signed infinity, with the exception of NaN, described above, and zero, described below.

Infinity and zero - The result of the multiplication of infinity and zero is undefined. The multiplier will therefore return a predefined NaN. If none of these cases apply, the special case path signals that the result of the standard path should be chosen.

V. RESULTS

This design has been implemented, simulated on ModelSim and synthesized for VHDL. Simulation based verification is one of the methods for functional verification of a design. In this method, test inputs are provided using standard test benches. The test bench forms the top module that instantiates other modules. Simulation based verification ensures that the design is functionally correct when tested with a given set of inputs. Though it is not fully complete, by picking a random set of inputs as well as corner cases, simulation based verification yield reasonably good results.

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.

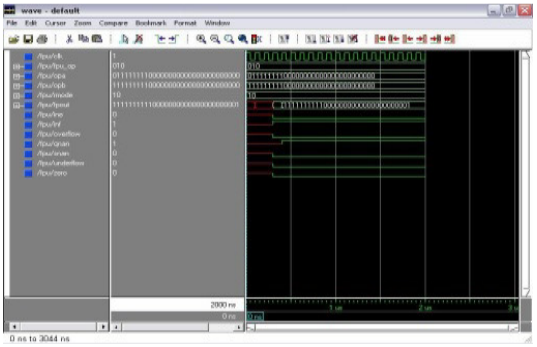

Figure 3: Output of Single Precision Floating Point Multiplier when above inputs aregiven

VI. Conclusion

Single precision floating point multiplier is designed and implemented using ModelSim in this thesis. The designed multiplier conforms to IEEE 754 single precision floating point standard. In this implementation exceptions (like invalid, inexact, infinity, etc) are considered. In this implementation rounding modes like round to positive infinity, round to negative infinity, round to zero and round to even. The designed is

verified using FPU test bench. The design is also verified for overflow and underflow cases.

References

1. John L Hennesy & David A. Patterson Computer Architecture A Quantitative Approach Second edition; A Harcourt Publishers International Company

2. J. Bhasker, A VHDL Primer, Third Edition, Pearson, 1999.

3. M. Ercegovac and T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004

4. John. P. Hayes, ìComputer Architecture and Organizationî, McGraw Hill, 1998.

5. Peter J. Ashenden, The Designerís Guide to VHDL, Morgan Kaufmann Publishers, 95 Inc., 1996.

6. Behrooz Parhami, Computer Arithmetic, Algorithms and Hardware Design Oxford University Press.2000.