A Single Precision Floating Point Multiplier for Machine Learning Hardware Acceleration

Devadasu Keerthana [1], M.Tech Research Scholar, Dept. of ECE, Eluru College of Engineering and Technology, JNTUK, AP

N. Chandra Paul [2], Assistant Professor, M.Tech, Dept. of ECE, Eluru College of Engineering and Technology, JNTUK, AP

A floating-point unit (FPU) colloquially is a math coprocessor, which is a part of a computer system specially designed to carry out operations on floating point numbers [1]. Typical operations that are handled by FPU are addition, subtraction, multiplication and division. The aim was to build an efficient FPU that performs basic as well as transcendental functions with reduced complexity of the logic used reduced or at least comparable time bounds as those of x87 family at similar clock speed and reduced the memory requirement as far as possible. The functions performed are handling of Floating-Point data, converting data to IEEE754 format, perform any one of the following arithmetic operations like addition, subtraction, multiplication, division and shift operation and transcendental operations like square Root, sine of an angle and cosine of an angle. All the above algorithms have been clocked and evaluated under Spartan 3E Synthesis environment. All the functions are built by possible efficient algorithms with several changes incorporated at our end as far as the scope permitted. Consequently, all of the unit functions are unique in certain aspects and given the right environment (in terms of higher memory or say clock speed or data width better than the FPGA Spartan 3E Synthesizing environment) these functions will tend to show comparable efficiency and speed, and if pipelined then higher throughput. . The booth-2 algorithm is used for the encoding operation, and the Wallace tree structure is used to complete the accumulation of partial products. A 32-bit single precision floating-point multiplier based on IEEE754 standard is designed. The multiplier can be used as the basic structure of hardware multiplier to accelerate Convolutional Neural Networks algorithm.

Index Terms:
Dadda, Floating Point, Multiplier, FPGA and Carry Save Multiplier.Dadda, Floating Point, Multiplier, FPGA and Carry Save Multiplier.

I. INTRODUCTION

Floating-point units (FPU) colloquially are a math coprocessor which is designed specially to carry out operations on floating point numbers [1]. Typically, FPUs can handle operations like addition, subtraction, multiplication and division. FPUs can also perform various transcendental functions such as exponential or trigonometric calculations, though these are done with software library routines in most modern processors. Our FPU is basically a single precision IEEE754 compliant integrated unit. In this chapter we have basically introduced the basic concept of what an FPU is, in the section 1.2. Following the section, we have given a brief introduction to the IEEE 754standards in section 1.3. After describing the IEEE 754 standards, we have explained the motivation and objective behind this project in section 1.4. And finally, the section 1.5contains the summary of the chapter.

FLOATING POINT UNIT

When a CPU executes a program that is calling for a floating-point (FP) operation, there are three ways by which it can carry out the operation. Firstly, it may call a floating-point unit emulator, which is a floating-point library, using a series of simple fixed-point arithmetic operations which can run on the integer ALU. These emulators can save the added hardware cost of a FPU but are significantly slow. Secondly, it may use an add-on FPUs that are entirely separate from the CPU, and are typically sold as an optional add-on which are purchased only when they are needed to speed up math-intensive operations. Else it may use integrated FPU present in the system [2]. The FPU designed by us is a single precision IEEE754 compliant integrated unit. It can handle not only basic floating-point operations like addition, subtraction, multiplication and division but can also handle operations like shifting, square root determination and other transcendental functions like sine, cosine and tangential function.

IEEE 754 STANDARDS

IEEE754 standard is a technical standard established by IEEE and the most widely used standard for floating-point computation, followed by many hardware (CPU and FPU) andsoftware

Copyright @ 2022 Author

implementations [3]. Single-precision floating-point format is a computer number format that occupies 32 bits in a computer memory and represents a wide dynamic range of values by using a floating point. In IEEE 754-2008, the 32- bit with base 2 format is officially referred to as single precision or binary32. It was called single in IEEE 754-1985. The IEEE 754 standard specifies a single precision number as having sign bit which is of 1 bit length, an exponent of width 8 bits and a significant precision of 24 bits out of which 23 bits are explicitly stored and 1 bit is implicit 1. Sign bit determines the sign of the number where 0 denotes a positive number and 1 denotes a negative number. It is the sign of the mantissa as well. Exponent is an 8 bit signed integer from −128 to 127 (2's Complement) or can be an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 single precision definition. In this case an exponent with value 127 represents actual zero. The true mantissa includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the mantissa appear in the memory format but the total precision is 24 bits.

For example:

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF

31 30 23 22 0

IEEE754 also defines certain formats which are a set of representation of numerical values and symbols. It may also include how the sets are encoded. The standard defines [4]:

• Arithmetic formats which are sets of binary and decimal floating-point numbers, which consists of finite numbers including subnormal number and signed zero, a special value called "not a number" (NaN) and infinity.

• Interchange formats which are bit strings (encodings) that are used to exchange a floating-point data in a compact and efficient form.

• Rounding rules which are the properties that should be satisfied while doing arithmetic operations and conversions of any numbers on arithmetic formats.

• Exception handling which indicates any exceptional conditions (like division by zero, underflow, overflow, etc.) occurred during the operations. The standard defines the following five rounding rules:

• Round to the nearest even which rounds to the nearest value with an even (zero) least significant bit.

• Round to the nearest odd which rounds to the nearest value above (for positive numbers) or below (for negative numbers)

• Round towards positive infinity which is a rounding directly towards a positive infinity and it is also called rounding up or ceiling.

• Round towards negative infinity which is rounding directly towards a negative infinity and it is also called rounding down or floor or truncation. The standard also defines five exceptions, and all of them return a default value. They all have a corresponding status flag which are raised when any exception occurs, except in certain cases of underflow. The five possible exceptions are:

• Invalid operation are like square root of a negative number, returning of qNaN by default, etc., output of which does not exist.

• Division by zero is an operation on a finite operand which gives an exact infinite result for e.g., 1/0 or log (0) that returns positive or negative infinity by default.

• Overflow occurs when an operation results a very large number that can't be represented correctly i.e. which returns ±infinity by default (for round-to nearest mode).

• Underflow occurs when an operation results very small i.e. outside the normal range and inexact (denormalised value) by default.

• Inexact occurs when any operation returns correctly rounded result by default.

II. LITERATURE REVIEW

When a CPU is executing a program that calls for a FP operation, a separate FPU is called to carry out the operation. So, the efficiency of the FPU is of great importance. Though, not many have had great achievements in this field, but the work by the following two are appreciable. Open Floating-Point Unit – This was the open-source project done by

Copyright @ 2022 Author

Rudolf Usselmann [6]. His FPU described a single precision floating point unit which could perform add, subtract, multiply, divide, and conversion between FP number and integer.

It consists of two pre-normalization units that can adjust the mantissa as well as the exponents of the given numbers, one for addition/subtraction and the other for multiplication/division operations. It also has a shared post normalization unit that normalizes the fraction part. The final result after post-normalization is directed to a valid result which is in accordance to single precision FP format. The main drawback of this model was that most of the codes were written in MATLAB and due to this it is non-synthesizable. GRFPU –This high Performance IEEE754 FPU was designed at Gaisler Research for the improvement of FP operations of a LEON based systems [7].

It supports both single precision and double precision operands. It implements all FP operations defined by the IEEE754 standard in hardware. All operations are dealt with the exception of denormalized numbers which are flushed to zero and supports all rounding modes. This advanced design combines low latency and high throughput. The most common operations such as addition, subtraction and multiplication are fully pipelined which has throughput of one CC and a latency of three CC. More complex divide and square root operation takes between 1 to 24 CC to complete and execute in parallel with other FP operations. It can also perform operations like converse and compliment. It supports all SPARC V8 FP instructions. The main drawback of this model is that it is very expensive and complex to implement practically.

III. PROPOSED METHOD

A transcendental function is a function whose coefficients are themselves polynomials and which does not satisfy any polynomial equation. In other words, it is a function that transcends the algebra in the sense that it is not able to express itself in terms of any finite sequence of the algebraic operations like addition, multiplication, and root extraction. Examples of this function may include the exponential function, the logarithm, and the trigonometric functions. In the approach of developing an efficient FPU, we have tried to implement some transcendental functions such as sine function, cosine and tangential functions. The operation involves usage of large memory storage, has large number of clock cycles and needs expensive hardware organization. To reduce the effect of the above-mentioned disadvantages, we have implemented CORDIC algorithm [13].

It is an effective algorithm to be used in our FPU as it can fulfil the requirements of rotating a real and an imaginary pair of a numbers at any angle and uses only bit-shift operations and additions and subtractions operation to compute any functions. Section 3 describes the efficient trigonometric algorithm using the CORDIC algorithm and brief introduction about the CORDIC function. The section further, describes the efficient trigonometric algorithm that was improvised to improve the operations of the FPU.

EFFICIENT TRIGONOMETRIC ALGORITHM

Evaluation of trigonometric value viz. sine, cosine and tangent is generally a complex operation which requires a lot of memory, has complex algorithms, and requires large number of clock cycles with expensive hardware organization. So usually it is implemented in terms of libraries. But the algorithm that we use here is absolutely simple, with very low memory requirements, faster calculation and commendable precision which use only bit-shift operations and additions and subtractions operation to compute any functions.

CORDIC FUNCTION

CORDIC (Co-ordinate Rotation Digital Computer algorithm) is a hardware efficient algorithm [14]. It is iterative in nature and is implemented in terms of Rotation Matrix. It can perform a rotation with the help of a series of incremental rotation angles each of which is performed by a shift and add/sub operation. The basic ideas that is incorporated is that –

- It embeds elementary function calculation as a generalized rotation step.
- Uses incremental rotation angles.
- Each of these basic rotation is performed by shift or and/sub operation Principles of calculation in figure 2 –

Copyright @ 2022 Author

- If we rotate point (1,0) by angle Ø then the coordinates say (X,Y) will beX= cos Ø and Y= sin Ø Ø and Y= sin Ø
- Now if we rotate (X.Y) we get say (X´, Y´), then it is expressed as X´= X.cos Ø – Y.sin Ø Y´= Y.cos Ø + X.sin Ø
- Rearranging the sameX´= cos Ø [X – Y. tan Ø]

Floating point multiplication is a crucial operation in high power computing applications such as image processing, signal processing etc. And also, multiplication is the most time and power consuming operation. This paper proposes an efficient method for IEEE 754 floating point multiplication which gives a better implementation in terms of delay and power.

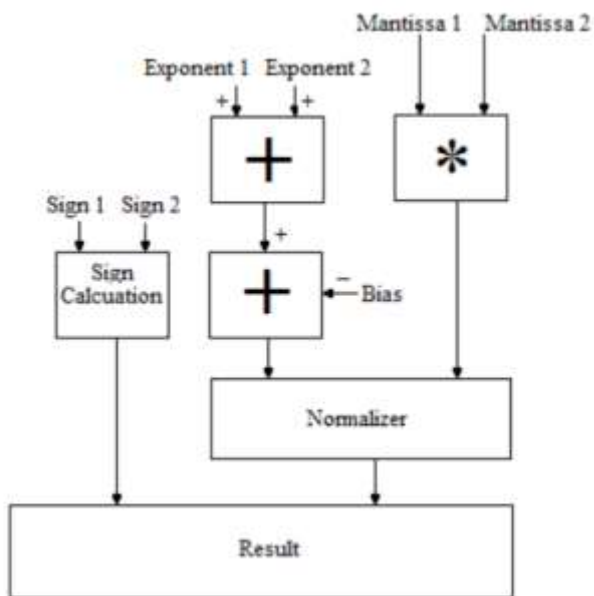

Figure1 Floating point multiplier

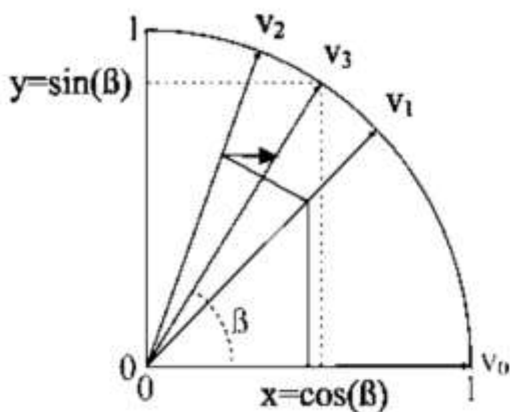

Figure 2 Cordic Angle Determination

Y´= cos Ø [Y + X. tan Ø] Where tan is calculated as steps tan Ø = ± 2-I The figure 2 describes the determination of the rotation angle by which the angles are determined to evaluate the trigonometric functions. The angle β used in the diagram is same as the angle Ø in the equations. So, basically CORDIC is an efficient algorithm where we would not prefer use of a hardware-based multiplier and we intend to save gates as in FPGA. Now, since our conventional input is in degrees, we built a look-up table in degrees. We are working towards a 12-bit precision structure. Moreover, since all our floating-point numbers have been converted to integers thus, we satisfy the criteria of fixed-point format. But since our calculations are all integer based, we need a look-up table that is integral in nature. So, we multiply the values in table by a value = 2048 (= 2 11 as we need a precision of 12 bits). Soour look-up table 1 is as follows-

Index	Ø	Ø * 2048
0	45	92160
1	26.565°	54395
2	14.036°	28672
3	7.125°	14592
4	3.576°	7824
5	1.789°	3664
6	0.895°	1833
7	0.4476°	917
8	0.2241°	459
9	0.1123°	230
10	0.0561°	115
11	0.0278°	57

Table 1 Look Up Table

We will assume a 12-step system so that it will yield 12 bits of accuracy in the final answer. Note that the Cos Ø constant for a 12-step algorithm is 0.60725. We also assume that the 12 values of Atan (1/2i) have been calculated before run time and stored along with the rest of the algorithm. If true FP operations are used then the shift operations must be modified to divide by 2 operations.

Copyright @ 2022 Author

INITIAL APPROACH:

The initialization specifies the total angle of rotation and sets the initial value of the point at(1,0) and multiplied by the constant 0.60725.

- Set register A to the desired angle.
- Set register Y to value 0
- Set register X to value 0.60725

Step 1: Set dx to value after shifting X right by i places (It effectively calculatesX*tan Ø for this step)

Step 2: Set dy to value after shifting Y right by i places (effectively calculates Y*tan Ø for this step)

Step 3: Set da to value Atan ($1/2^i$) (From the small lookup table)

Step 4: if value of A >= 0 (to decide if next rotation would be clockwise oranti-clockwise) then do,

Set value of X to value of X - dy (to compute X-Y*Tan Ø)

Set the value of Y to the value of Y + dx (To compute Y+X*Tan Ø) Set the value of A to the value of A - da (To update the current angle)

Step 5: if the values of A < 0 (to decide if next rotation would be clockwise or anti-clockwise) then do,

Set value of X to value of X + dy (to compute X-Y*Tan Ø)

Set the value of Y to the value of Y - dx (To compute Y+X*Tan Ø) Set the value of A to the value of A + da (To update the current angle)

Figure 3 Algorithm for CORDIC

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X. This algorithm requires the use of non-integral numbers. This presents certain inconvenience so the algorithm is modified to work with only integral numbers. The modified algorithm is given below. As we have been working with an algorithm using 12 bits, our output angle ranges from –2048 to +2047. So, we will have to assume 16-bit calculations throughout.

EFFICIENT CORDIC IMPLEMENTATION

- Set register A to the desired angle*2048
- Set register Y to value 0
- Set register X to the value of 0.60725*2048
- Setup the lookup table to contain 2048*Atan (1/2i)

COMPUTATION

The algorithm in figure 4 is described below. Do the following

Step 1: Set the value of dx to the value of after shifting X right by i places (done toeffectively calculate X*tan Ø)

Step 2: Set the value of dy to the value after shifting Y right by i places (doneeffectively to calculate Y*tan Ø)

Step 3: Set the value of da from the lookup ($1/2^i$) (From the small lookup table)

Step 4: if the value of A >= 0 (to decide if our next rotation is clockwise or anti clockwise), then do,

Set the value of X to the value of X – dy (to compute value of X-Y*Tan Ø) Set the value of Y to the value of Y + dx (to compute value of Y+X*Tan Ø)Set the value of A to the value of A – da (to update the current angle)

Step 5: if the value of A < 0 (to decide if our next rotation is clockwise or anticlockwise), then do,

Set the value of X to the value of X + dy (to compute value of X-Y*Tan Ø)Set the value of Y to the value of Y - dx (to compute value of Y+X*Tan Ø)Set the value of A to the value of A + da (to update the current angle) **Figure 4.3 Algorithm for efficient Trigonometric Evaluation**

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X. These outputs are within the integer range – 2048 to +2047.

IV. RESULTS AND DISCUSSION

In this chapter we analyse the results of simulation, RTL results and synthesis results for all the algorithms that we have implemented in our FPU.

Copyright @ 2022 Author

Then we compared the performance of our FPU to that of X87 family at similar clock speed. The synthesis was done in FPGA Spartan 3E Synthesizing Environment. The comparison is done with respect to

- Memory Requirement
- Gates Used
- Clock Cycle
- Complexity of the logic

SIMULATION RESULTS

The code was simulated in Xilinx 14.7. We have given some of the screen shots of the simulations that were obtained as a result of simulation in Xilinx software.

FLOAT TO INTEGER CONVERSION

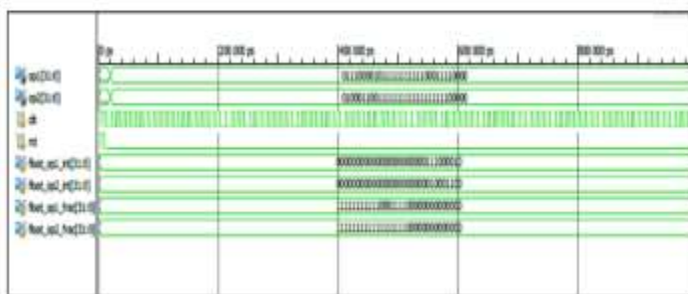

Figure 5 Float to Integer Conversion simulation result

The figure 5 gives the simulation result of float to integer conversion. The inputs are two 32bit operands, one for integral part and the other is fractional part. The output is the novel integral form of the input operands.

ADDITION

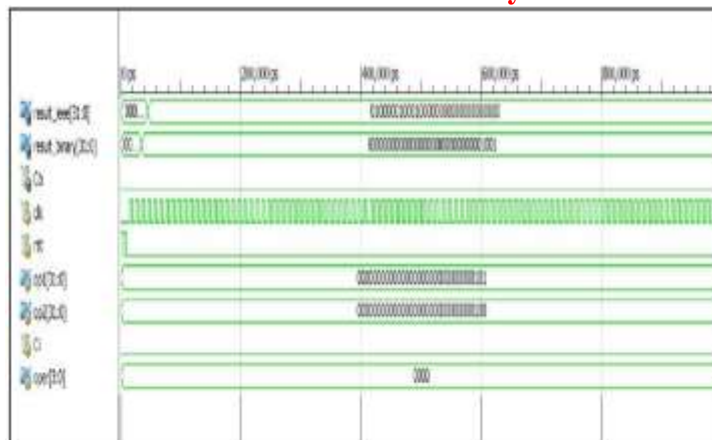

Figure 6 ADD simulation result

Figure 6 shows the simulation result of integer to IEEE format conversion. The input is the integer operand which was the output of the binary to integer representation conversion. The output is the IEEE representation of the input operand. The round mode is 00 and the operation mode is 0000.

SUBTRACTION

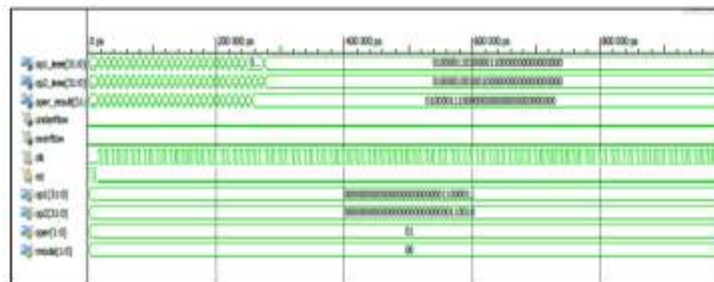

Figure 7 SUB simulation result

Figure 7 shows the simulation result for the subtraction operation. The input is the operands in the IEEE format and the output shows the resultant of the subtraction operation. The result also shows any exception encountered during the operation. The rounding mode is 00 and the operation mode is 0001.

MULTIPLICATION

Copyright @ 2022 Author

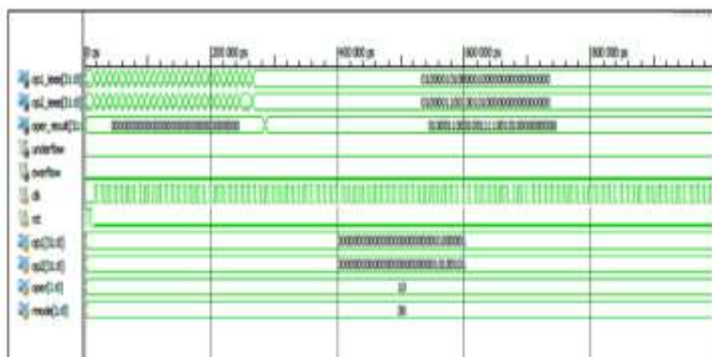

Figure 8 Multiplication simulation result

Figure 8 shows the simulation result for the multiplication operation. The input is the operands in the IEEE format and the output shows the resultant of the multiplication operation. The result also shows any exception encountered during the operation. The rounding mode is 00 and the operation mode is 0010.

DIVISION

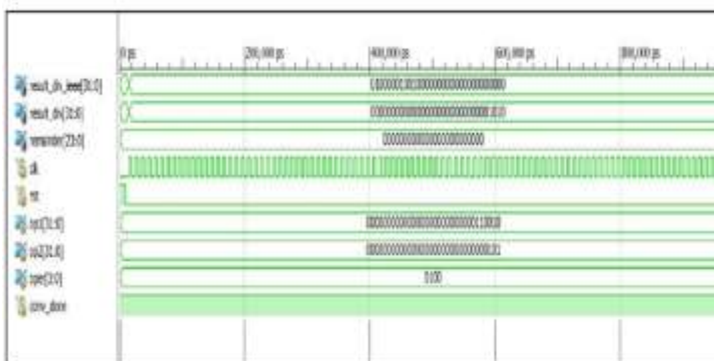

Figure 9 Division simulation result

Figure 9 shows the simulation result for the division operation. The input is the operands in the IEEE format and the output shows the resultant of the division operation. The result also shows any exception encountered during the operation. The rounding mode is 00 and the operation mode is 0100.

SHIFTING

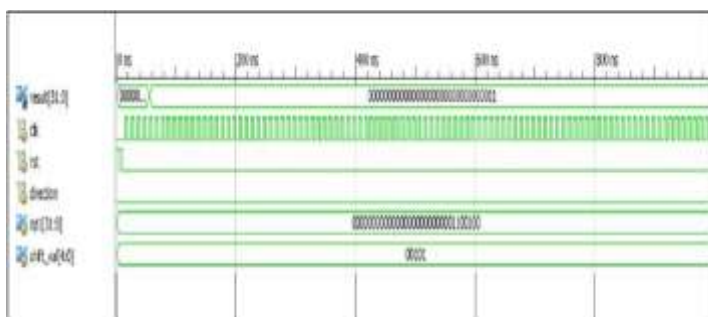

Figure 10 Shifting simulation result

Figure 10 shows the simulation result for the shifting operation. The input is the operands in the IEEE format and the output shows the resultant of the shifting operation. The result also shows any exception encountered during the operation. The rounding mode is 00.

SQUARE ROOT DETERMINATION

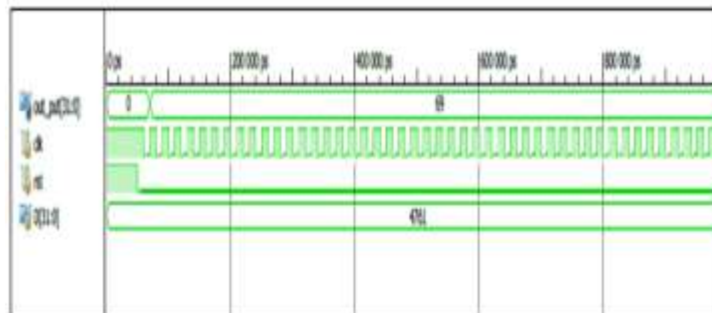

Figure 11 Square root simulation result

Figure 11 shows the simulation result for the square root determination operation. The input is the operands in the IEEE format and the output shows the resultant of the square root operation. The result also shows any exception encountered during the operation. The rounding mode is 00 and the operation mode is 0011.

TRIGONOMETRIC EVALUATION

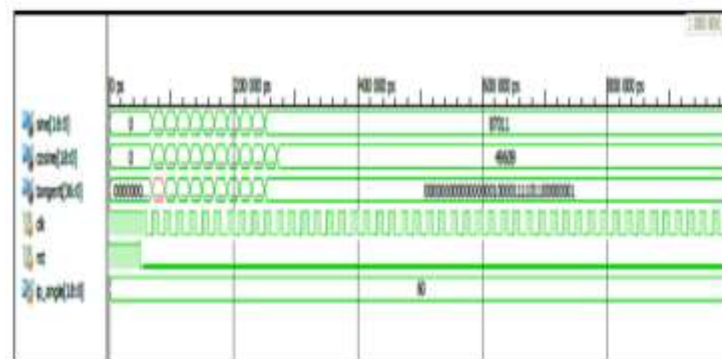

Figure 12 Trigonometric simulation result

Figure 12 shows the simulation result for the trigonometric operation. The input is the operands in the IEEE format and the output shows the resultant of the trigonometric operation. The result also shows any exception encountered during the operation. The rounding mode is 00 and the operation mode is 0110.

SYNTHESIS RESULTS

Copyright @ 2022 Author

After the simulation of the code was successful, we proceeded for the synthesis analysis. The simulation results gave a detailed description of the memory usage of the operation, i.e., the total number of registers required, total gates used, total multiplexers, LUTs, adders/subtractors, latches, comparator, flip--flops used. It also gives a detailed description of the device utilization summary, and detailed timing report which consists of time summary, timing constraints and delay. These details of the initial algorithm used which were discussed in chapter 2 were compared with that of the efficient algorithms discussed in chapter 3 and found that the efficient algorithms used less registers and gates. Number of IOs used was less in efficient algorithms and the delay were reduced too. For an example in table 2, the addition algorithm which was implemented using block CLA adder with reduced fan-in was using less number of gates and registers than used by normal block CLA and delay was also reduced in CLA with reduced fan-in.

	Block CLA	Block CLA with reduced Fan-in
1 Bit Register	52	28
24-Bit Register	2	3
Flip-Flops	100	70
1 Bit XORs	24	2
24-Bit XORs	1	2
Number of IOs	136	96
Delay (ns)	8.040	4.734

Table 2 Block CLA Vs. Block CLA with reduced fan-in

The synthesis report shows that the CLA with reduced fan-in is much more efficient than the normal CLA block algorithm. Thus proving the efficiency of the FPU designed. According to the simulation and synthesis results, we have compared the performance of our FPU with that of X87 family (PENTIUM/MMX). The following table 3 shows the result of comparison.

FPU	MAX CLK FREQ	DATA WIDTH	FADD /FSUB	FMUL	FDIV	FSQRT	FSIN /FCOS
PENTIUM /MMX	160-300 MHz	8 bit	1-3	1-3	39-40	70	17-173
OUR FPU (12 bit precsion)	50-250 MHz	32 bit	2-3	2-3	72	75-80	31

Table 3 OUR FPU Vs. PENTIUM/MMX

V. CONCLUSION

We have proved in the last chapter that the performance of our FPU was comparable to that of the X87 family (PENTIUM/MMX). The algorithm that we have used for the final FPU was comparable or even better in some case than the already existing efficient algorithms likein the case of block CLA and CLA with reduced fan-in in terms of memory used, delay, and device utilization. Because we have built the FPU using possible efficient algorithms with several changes incorporated at our ends as far as the scope permitted, all the unit functions are unique in certain aspects and given the right environment (in terms of higher memory or clock speed or data width better than the FPGA Spartan 3E synthesizing environment), these functions will tend to show comparable efficiency and speed and if pipelined then higher throughput may be obtained.

FUTURE WORK

Tough we have succeeded to achieve small amount of success in improvising the FPU, i.e. as per the results of synthesis and simulation, we have proved that our FPU have less memory requirement, less delay, comparable clock cycle and low code complexity, but still we have a vast amount of work that can be put on this FPU to further improvise the efficiency of the FPU. We can further implement operations like Exponential functions and Logarithmic functions. Further implementing Pipelining for the above operations can further increase the efficiency of the FPU. We also can encompass further exception logics like snan, qnan, ine, etc. We can also implement the FPU in Double precision format. Further, this code can serve as a skeleton for development of fault tolerant FPU at an exceedingly higher level.

REFERENCES

Copyright @ 2022 Author

[1] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago, "Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM", University of Southern California/InformationSciences Institute

[2] Christian Jacobi, "Formal Verification of a Fully IEEE Compliant Floating Point Unit", April 2002, Universit¨at des Saarlandes.

[3] Rudolf Usselmann, "Open Floating Point Unit, The Free IP Cores Projects".

[4] Edvin Catovic, Revised by: Jan Andersson, "GRFPU – High Performance IEEE754Floating Point Unit", Gaisler Research, Första Långatan 19, SE413 27 Göteborg, and Sweden

[5] David Goldberg, "What Every Computer Scientist Should Know About Floating Point Arithmetic", ACM Computing Surveys, Vol 23, No 1, March 1991, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

[6] Taek-Jun Kwon, Jeff Sondeen, Jeff Draper, "Design Trade-Offs in Floating-Point Unit, Implementation for Embedded and Processing-In-Memory Systems", USC Information Sciences Institute, 4676 Admiralty Way Marina del Rey, CA 90292 U.S.A.

[7] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago, "Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM", 2005, University of SouthernCalifornia/Information Sciences Institute [8] Yu-Ting Pai and Yu-Kumg Chen, "The Fastest Carry Lookahead Adder",

Department of Electronic Engineering, Huafan University [9] David Narh Amanor, "Efficient Hardware Architectures for Modular Multiplication", Communication and Media Engineering, February, 2005, University of Applied Sciences Offenburg, Germany/

[10] Andr¶e Weimerskirch and Christof Paar, "Generalizations of the Karatsuba Algorithm for Efficient Implementations", Communication Security Group, Department of Electrical Engineering & Information Sciences, Ruhr-UniversitÄat Bochum, Germany

[11] Yamin Li and Wanming Chu, "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations", International Conference on Computer Design (ICCD‟96), October 7–9, 1996, Austin, Texas, USA

[12] Claude-Pierre Jeannerod, Herv´e nochel, Christophe Monat, Member, IEEE, and Guillaume Revy, "Faster floating-point square root for integer processors", Laboratoire LIP (CNRS, ENSL, INRIA, UCBL)

[13] Prof. Kris Gaj, Gaurav, Doshi, Hiren Shah, "Sine/Cosine using CORDIC Algorithm"

[14] Samuel Ginsberg, "Compact and Efficient Generation of Trigonometric Functions usinga CORDIC algorithm", Cape Town, South Africa

[15] J. Duprat and J. M. Muller, "The CORDIC Algorithm: New Results for fast VLSIImplementation", IEEE Transactions on Computers, vol. C-42, pp. 168 178, 1993

Copyright @ 2022 Author