Development of Fixed-point Simultaneous Square Root and Logarithm Operation for High-level Synthesis

Kazunari Yoshikawa^{a*}, Naohiro Iwanaga^{a*}, Tatuya Hamachi^a, Akira Yamawaki^a

^aKyushu Institute of Technology, 1-1 Sensui-cho Tobata-ku Kitakyushu-shi Fukuoka-ken, 804-8550, Japan

*Corresponding Author: yama@elcs.kyutech.ac.jp

Abstract

When developing a hardware module performing some signal processing, we first design the algorithm in high level language like C. Next, we describe the behavior of the hardware module in hardware description language (HDL), referring to the C program previously written. Finally, a synthesis tool generates the real hardware from the HDL program. High-level synthesis technology (HLS) is a technology automatically converting C programs into the HDL programs. HLS can reduce the designing load of the hardware module by skipping the development of the HDL program. We are developing a fixed-point generic HLS library of elementary functions that is able to be converted by HLS. This paper focuses on simultaneous square root and logarithm operation in elementary functions. This paper shows that designing a fixed-point program in C language of simultaneous square root and logarithm operation by the coordinate rotation digital computer, CORDIC algorithm. To evaluate the hardware size and the operational speed of our operation, we convert it to the HDL program for an FPGA by Xilinx Vivado HLS tool. The experiment compares our hardware generated with the conventional floating point hardware generated from the math library. Although the conventional one is converted to the netlist dedicated to the specified device, the results show that our programs can decrease hardware size than the conventional one and achieve the equivalent performance. Our hardware library for HLS is the device-independent while the conventional one is converted to the netlist highly dependent on the specified device. Thus, our library have higher versatility than the conventional one.

Keywords: High-level synthesis, fixed-point, square root, logarithm, CORDIC

1. Introduction

Instead of the software implementation, the hardware implementation of some signal processing is often required to realize a high-performance embedded device.

However, the hardware design has high design load since it requires designing a high level language expressing the algorithm, developing an HDL program expressing the behavior of the hardware and performing a clock-cycle based operational verification. Thus, the high-level synthesis (HLS) technology automatically converting software in C language into the hardware in hardware description language (HDL) has been researched and developed [1].

Generally, signal processing such as audio processing, image processing uses elementary functions such as trigonometric, square root, logarithm functions, and so on. Conventional HLS implements elementary functions using floating-point number for keeping compatibility with math library of C language. However, the converted hardware of floating-point number function have problems from point of view such as processing time and power consumption, an increase of hardware size.

We are developing a generic elementary function library that is able to be HLS in fixed-point numbers [岩永論文]. This paper focuses on a simultaneous square root and logarithm operation in elementary functions. This paper attempts to design and develop the fixed-point program in C so that any HLS can generate the device-independent hardware module.

We employ coordinate rotation digital computer, CORDIC algorithm, to develop fixed-point simultaneous square root and logarithm operation. Operations CORDIC includes are suitable for hardware because they are

primarily additions and shift operations. Moreover, CORDIC is able to simultaneously realize various elementary functions such as trigonometric functions, exponential functions by changing some parameters and operations. When converting to hardware operations that use plural elementary functions, this feature indicates that an increase of hardware size is able to be suppressed for sharing of CORDIC modules.

However, the range of input value is narrow to use CORDIC correctly. Thus, our programs have to include a pre-process mapping an input value into the appropriate range and a post-process expanding the output value to the original range. So this paper shows a pre-process, a main part of fixed-point CORDIC algorithm and a post-process.

In the experiment, followings are performed. First, the appropriate range of input value for CORDIC is revealed. Next, the suitable number of repetitions is checked. CORDIC approximately obtains results by repeating the main loop. The number of repetitions increases the processing time. While, the smaller number of repetitions leads to the higher error. Their optimal parameters are investigated by making trade-off among the number of repetitions, the errors and the amount of hardware. Finally effectiveness of our square root and logarithm operations are shown by being compared with *sqr* and *log* functions about the error, the processing time and the amount of hardware.

2. Design of CORDIC

2.1 pre-process and post-process

This section describes the original CORDIC algorithm of square root and logarithm operations including pre-process and post-process. First, we describe the programs by CORDIC with floating-point. A square root is considered. The range where CORDIC can correctly execute has a lower bound and an upper bound. We think that the input value of the CORDIC square root is expressed by the form as shown in Eq. 1. The a is the input value. The $a * k^2$ is within the appropriate range for the CORDIC. We set k to 2. This is because the power of 2 can be realized by shift operation. This feature is desired in fixed-point.

$$k\sqrt{a} = \sqrt{a * k^2} \quad (1)$$

Moreover, input value of the CORDIC logarithm is expressed by the form as shown in Eq.2. The $a * k^c$ is

within the appropriate range for the CORDIC. We set k to 2. This is because the power of 2 can be realized by shift operation.

$$\log a + c * \log k = \log(a * k^c) \quad (2)$$

The pre-process and the post process are performed as follows from these features. When the input value is less than the lower bound, the pre-process of square root multiplies the input value by 4 until its result enters within the range. Also, the pre-process of logarithm multiplies the input value by 2 until its result enters within the range. When the input value is more than the upper bound, the pre-process of square root divides the input value by 4 as well. Also, the pre-process of logarithm divides the input value by 2 as well. CORDIC algorithm is performed after their pre-process. After the CORDIC square root, the post-process of square root returns the calculated value into the original range by multiplication of 2 or division of 2. After the CORDIC logarithm, the post-process of logarithm returns the calculated value into the original range by adds of $\log 2$ or subtract of $\log 2$.

2.2 CORDIC algorithm

Fig. 1 shows a pseudo code of square root and logarithm operations programs by CORDIC. At the line of 1, input parameters are a and $mode$. When we use the square root, the $mode$ is 1. when we use the logarithm, the $mode$ is 2. At the lines of 2 to 13, the pre-process reforms the input value to give the CORDIC the appropriate value. The s is a lower bound and the t is an upper bound of the appropriate range for the CORDIC. At the lines of 2 to 6, the number of repetitions is selected c or d . the c is the number of repetitions for logarithm. The d is the number of repetitions for square root. At the lines of 7 to 9, the small input value is multiplied by 2 or 4 until the multiplied value becomes bigger than the s . Moreover, the number of multiplications is recorded into the m . Likewise, at the lines of 8 to 12, the large input value is divided by 2 or 4 until it becomes smaller than the t . Moreover, the number of divisions is recorded into the n .

Then, we perform square root and logarithm operations by CORDIC at the lines of 14 to 27. CORDIC has two modes that are rotation mode and vector mode. Changing their modes is able to compute various functions. Square root and logarithm operations are used vector mode. Square root and logarithm operations of vector mode are expressed

by Eq.3 to Eq.10.

$$x[0] = a + 1 \quad (3)$$

$$y[0] = a - 1 \quad (4)$$

$$z[0] = 0 \quad (5)$$

$$x[j+1] = x[j] + \sigma_j 2^{-j} y[j] \quad (6)$$

$$y[j+1] = y[j] + \sigma_j 2^{-j} x[j] \quad (7)$$

$$z[j+1] = z[j] - \sigma_j \operatorname{arctanh}[j] \quad (8)$$

$$\sigma_j = \begin{cases} 1, & y[j] < 0 \\ -1, & y[j] \geq 0 \end{cases} \quad (9)$$

$$o = \begin{cases} x/1.65632, & \text{square root} \\ z * 2, & \text{logarithm} \end{cases} \quad (10)$$

```

1      Input      a, mode
2      IF mode = 1 THEN
3          j = c
4      ELSE mode = 2 THEN
5          j = d
6      ENDIF
7      WHILE a < s DO
8          a = a * (mode * 2)
9          m = m + 1
10     WHILE a > t DO
11         a = a / (mode * 2)
12         n = n + 1
13     ENDWHILE
14     x = a + 1
15     y = a - 1
16     z = 0
17     FOR i = 1 TO j DO
18         IF y < 0 THEN
19             x = x + y / 2i
20             y = y + x / 2i
21             z = z - arctanh[i]
22         ELSE y >= 0 DO
23             x = x - y / 2i
24             y = y - x / 2i
25             z = z + arctanh[i]
26         ENDIF
27     ENDFOR
28     IF mode = 1 THEN
29         o = z * 2
30         o = o + ((n - m) * log(2))
31     ELSE mode = 2 THEN
32         o = x / 1.65632
33         o = o * 2n-m
34     ENDIF
35     Output      o

```

Fig. 1 The pseudo code in floating-point.

x and y are coordinates. z is an angle. j is the number of repetitions. σ_j is direction of vector rotation. o is output value. Moreover, $\operatorname{arctanh}$ that is hyperbolic arc tangent beforehand prepared for the number of repetitions.

At lines of 28 to 34, the post-process returns o within the original range. When we use the logarithm, it performs some add $\log 2$ multiplication of $(n - m)$ according to the recorded number in the m and n . When we use the square root, the post-process performs some multiplications or divisions according to the recorded number in the m and n .

3. Operations in Fixed-Point

This section shows algorithm of the fixed-point square root and logarithm operations. We convert the floating-point square root and logarithm operations into the fixed-point square root and logarithm operations. About allotment of bits in fixed-point, the integer part has p bits, the fractional part has p bits, and the number of bit shift in computation is r bits. Therefore, a must be made q bits left shift. Fig. 2 shows a pseudo code of program about square root and logarithm operations in fixed-point. At the lines of 7 to 13, a power of 2 is shift operations in fixed-point. Therefore, in the pre-process, a multiplication of 4 is carried out by 2 bit left shift and a division of 4 is carried out by 2 bit right shift. At the lines 14 to 27, CORDIC in floating point computation is converted into a fixed-point calculation. At the lines of 14 to 16, “1” used for the calculation is shifted to the q bit left. At the lines of 17 to 27, division of 2 is carried out by 1 bit right shift. At the line of 29, division of 2 is carried out by 1 bit right shift. At the line of 30, v is $\log 2$ in fixed-point. At the line of 32, u is value of $(1/1.65632 \ll r)$ in fixed-point. The post-process returns o within the original range. When we use the logarithm, it performs some add $\log 2$ multiplication of $(n - m)$ according to the recorded number in the m and n . When we use the square root, the post-process performs some multiplications or divisions according to the recorded number in the m and n . However, o is performed r bits left shift in line of 32. Therefore, the post-process must perform r bits right shift. Accordingly, square root and logarithm operations are able to compute with fixed-point.

```

1      Input      a, mode
2      IF mode = 1 THEN
3          j = c
4      ELSE mode = 2 THEN
5          j = d
6      ENDIF
7      WHILE a < s DO
8          a = a << mode
9          m = m + 1
10     WHILE a > t DO
11         a = a >> mode
12         n = n + 1
13     ENDWHILE
14     x = a + (1 << q)
15     y = a - (1 << q)
16     z = 0
17     FOR i = 1 TO j DO
18         IF y < 0 THEN
19             x = x + (y << i)
20             y = y + (x << i)
21             z = z - arctanh[i]
22         ELSE y >= 0 DO
23             x = x - (y << i)
24             y = y - (x << i)
25             z = z + arctanh[i]
26         ENDIF
27     ENDFOR
28     IF mode = 1 THEN
29         o = z << 1
30         o = o + ((n - m) * v)
31     ELSE mode = 2 THEN
32         o = x * u
33         o << n
34         o >> (m + r)
35     ENDIF
36     Output      o

```

Fig. 2 The pseudo code in fixed-point.

4. Experiment and Discussion

4.1 Input value range

This section shows the range of input value of CORDIC. Then, we perform square root and logarithm operations range of 1.0-40.0 that don't perform the pre-process and the post-process for determination of input value range for CORDIC. We performed the error comparison with *sqrtf* function of math library. We performed the error comparison with *logf* function of math library. Fig. 3 shows this result of square root. Fig. 4 shows this result of logarithm. As a result, the error is the minimum in range 2.0 to 8.0. This paper adopts this range. As a result, the error is the minimum in range 4.0 to 8.0. This paper adopts this range.

Fig. 3 the error comparison with *sqrtf* function

Fig. 4 the error comparison with *logf* function

4.2 Number of repetitions

This section shows the number of repetitions of CORDIC. At the repetitive operation for CORDIC by Eq. 6 to Eq. 9, we searched the maximum error of our function that compare math library (m-lib), the number of repetitions increase one by one from 0 determine for the appropriate number of repetitions. The maximum error of square root saturated that the number of repetitions is three or more times. Also, the maximum error of three is 0.938 (%) that is a practical error. The maximum error of logarithm saturated that the number of repetitions is six or more times. Also, the maximum error of six is 3.694 (%) that is a practical error.

4.3 Number of clocks

This section shows the appropriate number of clocks. We performed the logic simulation in 3 to 8 times in the

Fig. 5 Compare of the number of clocks.

number of repetitions to ascertain the number of clocks. Fig. 5 shows this result and *sqrtf* and *logf* function result of math library

.As the result shown in Fig. 5, as the number of clocks decreases the number of repetitions decreases. Also, the minimum number of clocks is 12 when the number of repetitions is three. This is the same as the value 12 that the number of clocks of math library. Other the number of repetitions is bigger than 12. However, its difference is few.

4.4 Implementing Experiment on FPGA

This section shows the hardware size. We performed HLS to our fixed point square root and logarithm with the number of repetitions from 3 to 8 by Xilinx Vivado HLS tool to Xilinx FPGA. Used FPGA is *zynq*. In addition, we performed HLS and implemented at FPGA for the floating point *sqrtf* and *logf* of the math library as a compared target. Fig. 6 shows the result about registers used. Also, Fig. 7 shows the result about LUTs used. As the result shown in Fig. 6, registers used of our operations decrease 36% compared with math library when the number of repetitions is 6. The result of Fig. 7 indicates ythat LUTs used of our operations decrease 14% compared with math library when the number of repetitions is 6. Therefore, our hardware can decrease the hardware size compared with math library.

Fig. 6 Comparison of registers used.

Fig. 7 Comparison of LUTs used.

5. Conclusions

We developed C program of fixed-point square root and logarithm operations.

Since CORDIC is used, the range of an input value is narrow. According to the basic experiment, we proved that the error with *sqrtf* and *logf* functions of math library is a practical error. When we use the square root, input value of CORDIC is range of 2.0 to 8.0. When we use the logarithm, input value of CORDIC is range of 4.0 to 8.0. Therefore, we include the pre-process and the post-process. Moreover, CORDIC obtains approximate results by repeating the main loop. When this number of repetitions is five or more times, we proved that the above error is kept.

We showed that hardware size decrease 36% compared with math library. In addition, it uses the net list when using math library. It means that it can be used only by a specific device. Compared with this, our program is the versatility because it doesn't use the net list. From this result, we developed highly useful hardware of fixed-point square root operations which is able to be converted by HLS.

As future work, we will develop something programs

using the square root or logarithm operation.

References

- (1) Wim Meeus, Kristof Van Beeck, Toon Goedemé , Jan Meel, Dirk Stroobandt, “An overview of today’s high-level synthesis tools”, Design Automation for Embedded Systems, Volume 16, Issue 3 , pp 31-51, 2012.
- (2) Xilinx: “Xilinx Vivado Design Suite user’s guide: HLS-Xilinx”
http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf (accessed 2th, Dec, 2013)
- (3) Impulse “Impulse C ANSI-C math library support package Xilinx “Xilinx Vivado Design Suite user’s guide: HLS-Xilinx”
http://japan.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf
(accessed 11th, Dec, 2013)