

Datasheet of FP divisor and square root

Lei Li (*lile@iis.ee.ethz.ch*)

06/02/2017

1. Algorithms and functions

Divisor employs the non-restoring binary divisor algorithm (NRBD)(K. Jun, and E. E.Swartzlander, Modified non-restoring division algorithm with improved delay profile and error correction, IEEE). And square root uses the non-restoring square root calculation algorithm (NRSC)(Y. Li, and W. Chu, Implementation of single precision floating square root on FPGAs,IEEE). To reduce the area overhead, they are designed with one shared control logic and share the used iteration cells. For divisors, to improve the accuracy, an extra MSC and adder is added (K. Jun, and E. E.Swartzlander, Modified non-restoring division algorithm with improved delay profile and error correction, IEEE).

Both support IEEE 754 for single precision. The three basic components in IEEE 754 are sign(S), exponent(E) and mantissa(M).

Table 1 IEEE 754 for single precision

| Precision | Sign | Exponent | Mantissa | Bias |
|-----------|-------|----------|----------|------|
| Single | 1[31] | 8[30:23] | 23[22:0] | 127 |

$$= (-1)^S \times (1.M)^E$$

Table 2 Special bit patterns in IEEE 754

| | M=0 | M≠0 |
|-------|--------------|-----------------------------------|
| E=0 | 0 | Denormalized with real exponent=1 |
| E=255 | $\pm \infty$ | NaN |

The IEEE 754 2008 standard supports all floating point operations. It handles all special inputs including signaling NaN, quiet NaN,+Infinity,-Infinity, positive zero and negative zero. Our design fully supports IEEE 754 and offers three exceptions namely overflow(OF), underflow(UF) and division by zero(DZ). Besides, our design supports four different rounding modes: RNE(Round to Nearest, ties to Even, 00), RTZ(Round towards Zero, encoding as 01), RDN(Round Down, encoding as 10), RUP(Round Up, encoding as 11).

This document is organized as follows: Chapter 2 will summarize all inputs/outputs. Chapter 3 will introduce the architecture. Chapter 4 will address normalization. Chapter 5 will show the rounding modes. Chapter 6 will present exceptions. Chapter 7 will provide some waveforms for simulations. Chapter 8 will give the synthesized results.

2. Inputs and Outputs

div_sqrt_top is the name of our design, which can be used to divide two floating point operands: Operand_a_DI by Operand_b_DI to produce a floating-point quotient, Result_DO, and compute the floating-point square root of a floating-point operand, Operand_a_DI. The input RM_SI is a 2-bit rounding mode.

Table 3 Inputs and outputs

| or | width | direction | Function |
|---------------|--------|-----------|---|
| Clk_CI | 1 | IN | Clock |
| Rst_RBI | 1 | IN | Reset, active low |
| Div_start_SI | 1 | IN | Start the operation of divisor. Active high for one cycle. |
| Sqrt_start_SI | 1 | IN | Start the operation of square root. Active high for one cycle. |
| Operand_a_DI | 32bits | IN | Div: Numerator; Sqrt:Radicand |
| Operand_b_DI | 32bits | IN | Div: Denominator |
| RM_SI | 2 bits | IN | Rounding mode. |
| Result_DO, | 32bits | OUT | Div: Quotient with one cycle ; Sqrt: Square root of Operand_a_DI with one cycle |
| Done_SO | 1 | OUT | Active high for one cycle |
| Ready_SO | 1 | OUT | Active high. It will hold high state until the next Div_start_SI or Sqrt_start_SI arrives |

3. Architecture

According to radix 2 ($r=2$) NRBD and NRSC, n iterations are needed for n -bit operands. For IEEE single precision, 24 iterations are needed with 23-bits mantissa and 1 hidden bit. 24 iterations can be implemented using an iteration unit with 24 cycles, or using m iteration units with $24/m$ cycles. Thus, the appropriate m should be chosen.

For division, each iteration can be seen to be same and the control logic is comparatively simple. On-the-fly conversion and an extra MSC and adder are used to produce the final

quotient. The key point of the control logic is how to store the generated quotients each cycle and how to select the needed quotient to choose the appropriate operands at the first iteration unit. An efficient method is to shift the quotient registers by 24/m each cycle. Thus we can use a fixed register to choose.

The control logic of square root is more complex than that of divisor. It is because each iteration of square root is different with different intermediate operands. We have to add some fine-grained control. The corresponding selectors are controlled by a finite state machine (FSM).

The employed architecture is shown in Fig.1. *div_sqrt_top* is the top module, which is consisted of three modules: *preprocess*, *nrbd_nrsc* and *fpu_norm*. *nrbd_nrsc* contains a control logic and four iteration units. The design can be seen as three stages: the first stage, the middle stage and last stage. To reach the target clock period of 2.8ns, using UMC65nm process technology, the solution based on four iteration units at the middle stage is chosen. $8(=1+24/4+1)$ cycles are needed for producing the final results. The first cycle is used to store operands and generate control signals at the first stage. The 2nd-7th cycles are used to finish 24 iterations at the middle stage. The 8th cycle is used to normalize and round the result. The output result is then ready at the last stage (without flip/flops). In other words, the output results can be captured at the rising clock edge of the 8th cycle. Big margins are kept for the inputs and outputs, about 1ns.

In the *preprocess* module, two operands are unpacked into two IEEE-754 encoded numbers into corresponding sign bits, biased binary exponents, and mantissa. To support denormal numbers, two leading zero detectors(LZD) are added to counter the number of leading zeros in mantissa part of both operands. With LZD1 and LZD2, two operands are normalized. The resultant exponent for division can be calculated by $(Exp_a_D - Exp_b_D + Bias + LZ2 - LZ1)$. For square root, the resultant exponent can be computed as

$$\frac{Exp_a_D - LZ1 - 127}{2} + 127 = \frac{Exp_a_D - LZ1}{2} + 63 + (Exp_a_D - LZ1)\%2$$

The result exponent and normalized operands are stored into flip/flops (*Exp_norm_D* and *Mant_norm_D*) for next stage. The sign of final result is calculated by using sign of both operands or one based on *Div_start_SI* and *Sqrt_start_SI* and stored into a flip/flop. Operand detection is added to generate *Inf_a_S*, *Inf_b_S*, *Zero_a_S*, *Zero_b_S*, *NaN_a_S* and *NaN_b_S* for normalization of the final result. For the special cases *NaN_a_S*=1 or *NaN_b_S*=1, the input operands are needed to store into flip/flops for normalization.

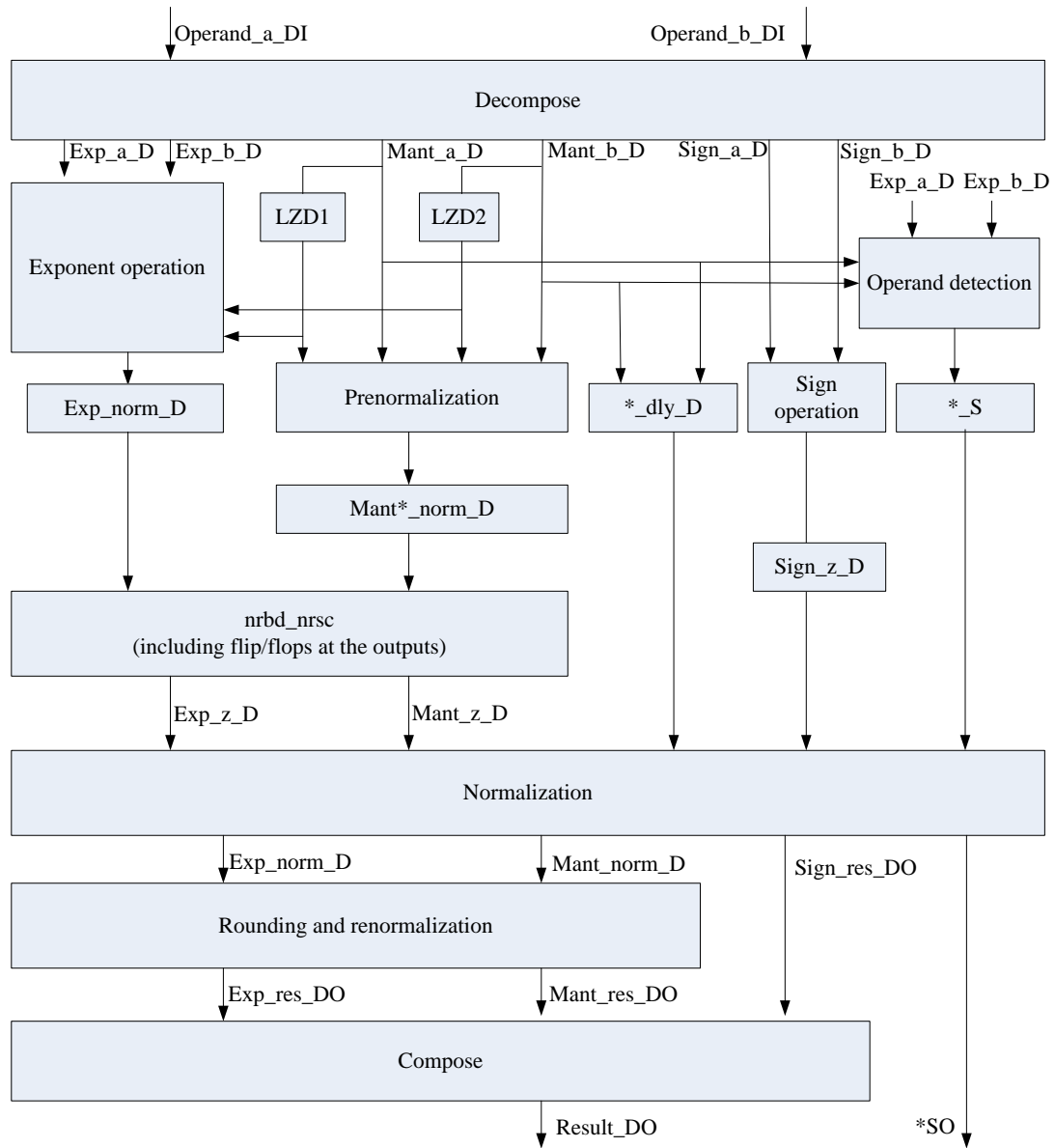


Fig.1 The architecture for the shared FP divisor and square root
 ** *_D or *_S in a block are flip/flops.

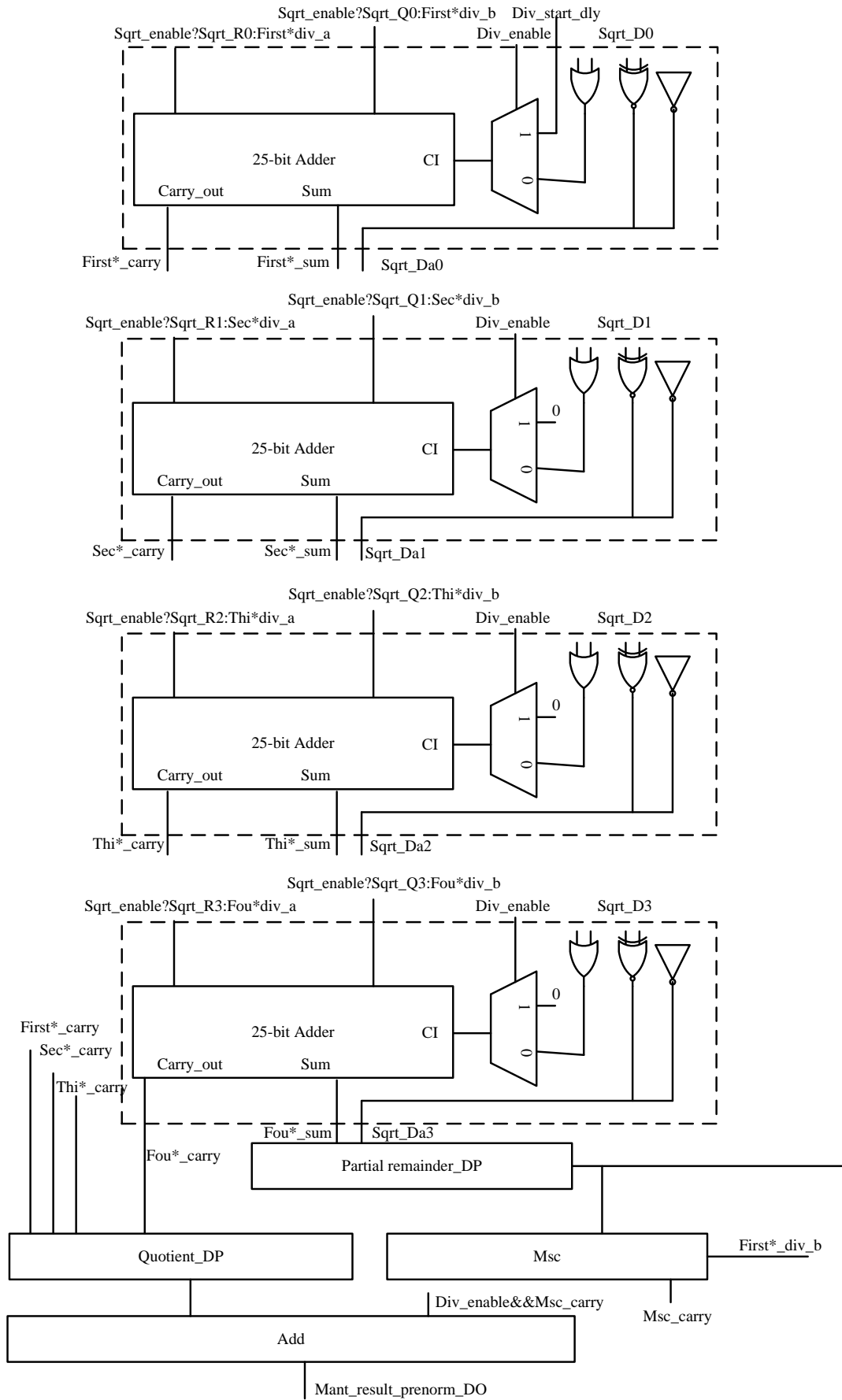


Fig.2 The data flow of the shared divisor and square root in *nrbd_nrsc*

Fig.2 shows the data flow in *nrbd_nrsc*. A finite state machine (000-101) is used to control it. *Sqrt_enable_S* is used to choose the operands for square root and division. For division, in the first iteration the left shift is not needed (Just like “the traditional pencil-and-paper method”, when we do a division, no left shift is needed before the first substraction. Besides, it breaks through the restriction $Mant_a_D < Mant_b_D$ for some divisors) and the 2’s complement introduced 1 should be added as carry-in in iteration_cell_for first by adding *Div_start_dly_SI*. *First*div_a* is chosen from *Mant_a_norm_D* from *preprocess* or *Partical_remiander_DP* based on *Div_start_dly_SI*. The other input **div_a* of next iteration cell are from Sum of the previous iteration cell directly, **_sum*. For example, the input *sec*div_a* of the second iteration cell is from *first*_sum*, the Sum of the first iteration cell. **div_b* is chosen from +denominator or – denominator according to the *Carry_out* of the previous iteration cell **_carry*. All the *Carry_outs* of iteration cells are stored into *Quotient_DP*. The final quotient should be *Quotient_DP[MANT-1:0]+Msc_carry*, shown in Fig. 2. Herein *Partical_remiander_DP* and *Quotient_DP* are flip/flops.

Square root is more complex than division. *Sqrt_D** is chosen from *Sqrt_mant_a_norm_D*, 2bits for each iteration. *Sqrt_R0* is chosen from ‘0 or *Partical_remiander_DP* based on *Sqrt_start_dly_SI*. The other input *Sqrt_R** of next iteration cell are from Sum(**_sum*) and *D_DO(Sqrt_Da*)* of the previous iteration cell directly. *Sqrt_Q** are different in each iteration with increase numbers, which are the *Carry_outs* of the finished iterations.

Quotient_DP[MANT-1:0] is the result of square root before normalization.

The control signal from instruction decoder are *Div_start_SI* and *Sqrt_start_SI*. They will be stored in flip/flops as *Div_start_dly_S* and *Sqrt_start_dly_S*, and be used to generate *Div_enable_S* and *Sqrt_enable_S* in control module.

Fpu_norm include normalization and rounding and renormalization. The employed schemes are shown in Section 4 and rounding in Section 5. The produced exception flags will be given in Section 6.

4. Normalization

4.1 division

(1)For normal IEEE754 operands, the result mantissa of division should start with 1 or 01. Therefore, we just need to care about the MSB of the quotient. When the quotient is 1.XXX, we check if the resultant exponent $Exp_a_D - Exp_b_D + bias$ is out of the range of exponent. When the quotient is 0.1XX, we need to shift the mantissa one bit to the left and correct the exponent to: $Exp_a_D - Exp_b_D + bias - 1$.

$$\frac{1.M1}{1.M2} = 1.XXX \text{ or } 0.1XX$$

(2) If the numerator is a denormal number,

$$\frac{0.M1}{1.M2}$$

The resultant exponent is $\text{Exp_a_D} - \text{Exp_b_D} + \text{bias}$. May be a negative number. Index of LZD should be checked for normalization. If it is a negative number, we have to right shift the mantissa by Index of LZD to check if the resultant exponent ($E + \text{Index of LZD}$) is negative. If it is negative, it is overflow.

(3) If the denominator is a denormal number,

$$\frac{1.M1}{0.M2} > 1$$

It is analyzed above.

(4) If these two operands are denormal numbers,

$$\frac{0.M1}{0.M2}$$

We should detect the first ones of these operands. It is the reason that we added two LZDs before operation.

Solution: If the hidden bit is 0, we should detect the first one of operands and left shift these two operands to normal mantissas. Thus we just care about exponent. $\text{Exponent} = \text{Exp_a_D} - \text{Exp_b_D} + \text{Bias} + \text{LZ2} - \text{LZ1}$, can be positive or negative. The leading one of the quotient should be detected for normalization.

(5) Other cases

If $1 \leq E \leq 254$, it is a normal result, return E and M;

If $E=0$ and $M \neq 0$, it is a denormal number, return $\gg (M)$ and the adjusted E;

If E is negative, the numbers cannot be represented. OF is signaled and return $E=0, M=0$ (If so, it is all right for testbench. If not, cannot pass the check);

If $E=255$ and $M \neq 0$, NaN is signaled and return $E=255, M=0$;

If $E > 255$, OF is signaled and return $E=255, M=0$.

(6) Special cases

Table 4 Special cases for division

| Division | operation | return |
|----------|----------------|---|
| 1 | a/NaN | The input NaN |
| 2 | a/Inf | 0, the sign depending on the two operands |
| 3 | $a/0$ | Inf, the sign depending on the two operands |
| 4 | $0/b$ | 0, the sign depending on the two operands |
| 5 | Inf/b | Inf, the sign depending on the two operands |
| 6 | NaN/b | The input NaN |
| 7 | $0/\text{Inf}$ | 0, the sign depending on the two operands |

| | | |
|---|-------|---|
| 8 | Inf/0 | Inf, the sign depending on the two operands |
|---|-------|---|

4.2 square root

(1) The operand is a normal number

For normal IEEE754 operands, the result mantissa of square root should start with 1. Thus, we can check the final exponent. The 1.M will be left shifted one or zero-bit so that the new exponent e' makes $e' - 127$ even. The shifted fraction will be 1X.XXX or 01.XXX. The result value will be 1.XXX. The resultant exponent can be computed as

$$\frac{e - 127}{2} + 127 = \frac{e}{2} + 63 + e\%2$$

(2) The operand is a denormal number

If the input operand is a denormal number, we can left shift like division. e can be a negative number. If e is even, it needs left shift 1-bit more.

(3) Other cases

Same to division.

(4) Special cases

Table 5 Special cases for square root

| Square root | operands | |
|-------------|----------|---------------|
| 1 | 0 | +0 |
| 2 | NaN | The input NaN |
| 3 | Inf | +Inf |

These special cases can be covered by division.

5. Rounding

The design supports four different rounding modes: RNE(Round to Nearest, ties to Even, 00), RTZ(Round towards Zero, encoding as 01), RDN(Round Down, encoding as 10), RUP(Round Up, encoding as 11).

Table 6 Rounding modes

| Mode | Code |
|------|------|
| RNE | 00 |
| RTZ | 01 |
| RDN | 10 |
| RUP | 11 |

6. Exceptions

UMC65nm process technology was used for synthesis.

Operating Conditions: uk65lsc1lmvbb1_108c125_wc

Library: uk65lsc1lmvbb1_108c125_wc

Input_delay:1ns

Output_delay:1ns

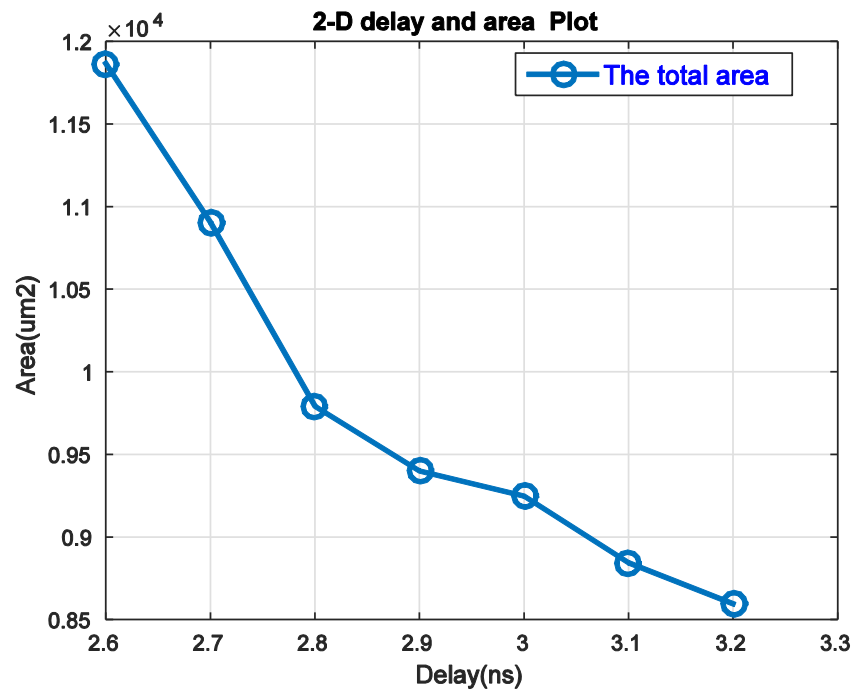


Fig.6 The synthesized results