EFFICIENT ON-LINE TESTING METHOD FOR A FLOATING-POINT ADDER

A. Drozd , M. Lobachev
Department of Computer Systems, Odessa State Polytechnic University, Odessa, Ukraine
Drozd@ukr.net, Lobachev@ukr.net

Abstract

In this paper we present a residue method for on-line testing of the floating-point adder. This circuit contains arithmetic shifter which executes an abridged operation. In the method the problem of the abridged operation checking with the reduced hardware amount is solved.

1. Introduction

The addition of numbers is the most spread arithmetic operation. It is simply executed and checked in fixed-point circuits. The residue checking is traditionally used [1]. In this method for number its check code – modulo residue is generated. The addition of the check codes followed by the residue calculation defines check code of the sum. The error indication code of sum is calculated by comparison the sum with its check code. The process of addition considerably becomes complicated in floating-point circuits [2]. In addition the common traits for arithmetic floating-point operations appear:

- The mantissa processing is executed with usage of multiplication.
- The result of this multiplication contains only n high bits of $2n$-bit product the n-bit multiplicands.
- In combinatorial circuits the multiplication is executed with abridged computation because of large hardware amount connected to operand size by the square dependence. The computation abridgement reduces the hardware amount almost twice without loss of accuracy.

The floating-point addition contains the multiplication as arithmetic shift of mantissa. This operation is executed with the computation abridgement reducing hardware amount twice in comparison with long shift.

Traditional application of residue checking for on-line testing of floating-point adder requires restoring the abridged operation to long shift. The offered method solves the problem of abridged operation checking with essential decrease of the hardware amount.

2. Floating-point addition

In this section an addition of floating-point numbers $A=a_1 \times 2^{a_2}$ and $B=b_1 \times 2^{b_2}$ is considered. The result $C=A+B$ is presented as $C=c_1 \times 2^{c_2}$. This operation is executed by a matching of exponents a_2 and b_2. In the process of exponent calculation the result order $c_2 = \max(a_2, b_2)$ and numbers $d_a = c_2 - a_2$, $d_b = c_2 - b_2$ are computed.

Then the mantissas a_1 and b_1 are shifted to down of d_a and d_b positions (one of the numbers d_a or d_b is equal to zero): $a_{1\ shift} = a_1 \times 2^{-d_a}$, $b_{1\ shift} = b_1 \times 2^{-d_b}$.

The result mantissa is calculated by the formula

$$c_1 = a_{1\ shift} + b_{1\ shift}.$$

A floating-point adder executing described operation is shown in Fig. 1. It consists of the block 1 for the exponent processing, arithmetic shifters 2 and 3, adder 4.

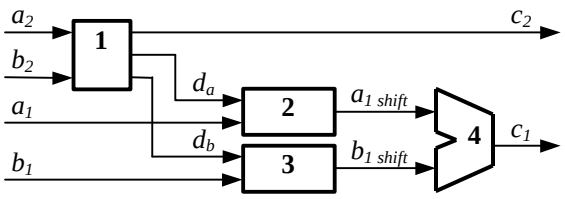

Figure 1: Floating-point adder

The block 1 calculates the result exponent c_2 and the numbers d_a, d_b. The blocks 2 and 3 execute the operations of an arithmetic shift of d_a and d_b positions for mantissa a_1 and b_1. The block 4 calculates the result mantissa c_1.

3. Arithmetic shift of a mantissa

In this section we consider the execution of arithmetic shift which is the most complex operation executed in the floating-point adder. The arithmetic shift to down of d positions for mantissa $a\{1,n\}$ is shown in Fig. 2.

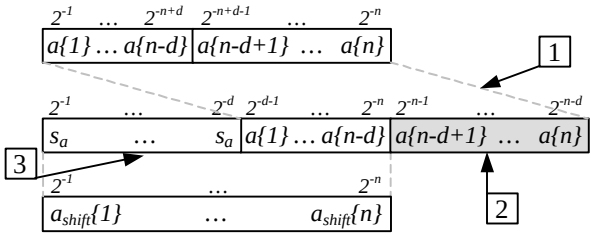

Figure 2: Arithmetic shift

The operation consists of three actions:

1) The reduction of the bit weights for the mantissa a in 2^d times.

2) The abridgement of the d low bits of the mantissa a. These bits compose the code $a_0=a\{n-d+1\,,n\}$.

3) The sign bit padding in the position with bit weights $2^{-1}\,,2^{-d}$ for complement code of the mantissa a. Sign bits $s_a \ldots s_a$ compose the code a_s.

It defines the arithmetic shift execution by the formula:

$$a_{shift} = a\,2^{-d} - a_0 + a_s. \qquad (1)$$

The mantissa arithmetic shift is a multiplication on 2^{-d} and executed with the computation abridgement on arithmetic shifter. This circuit is shown in Fig. 3 for $n=15$.

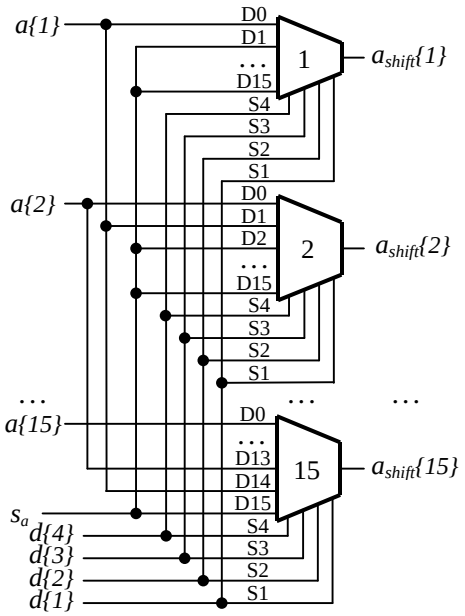

Figure 3: Arithmetic shifter

The circuit contains n of n-*to*-*1* multiplexers. Each multiplexer calculates one bit of the shifted mantissa a_{shift}. This bit is selected from the bits of the mantissa a and sign bit s_a under the control of the code d. The multiplexer hardware amount q is proportional to the operand size n. The arithmetic shift hardware amount $Q_{a.shift}=nq$ is proportional to the square of the operand size n and make main hardware amount $Q_{add}=2nq$ of the floating-point adder.

In table 1 is shown the shift matrix which describes arithmetic shift as the abridged operation**.**

The row of the shift matrix contain the various cases of the mantissa bit states after shift of $d=0\,,n$ positions (for $n=15$). The code $d\{1\,,r\}$ is presented in binary notation for $r=4$. The part a_0 is not calculated. The first 15 columns of the shift matrix describe functions executed on the multiplexers of the arithmetic shifter. The column contains the mantissa bits selected on the multiplexer output from its information inputs at appropriated values of the code d.

Table 1: Shift matrix

d	$a = a\{1\,,n\}$																														
4 3 2 1	1	2	3	4	…	12	13	14	15																						
2^3 2^2 2^1 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…	2^{-12}	2^{-13}	2^{-14}	2^{-15}																						
0 0 0 0	1	2	3	4	…	12	13	14	15																						
0 0 0 1		1	2	3	4	…	12	13	14	15																					
0 0 1 0			1	2	3	4	…	12	13	14	15																				
0 0 1 1				1	2	3	4	…	12	13	14	15																			
0 1 0 0					1	2	3	4	…	12	13	14	15																		
…								…			…			…																	
1 1 0 0								1	2	3	4	…	12	13	14	15															
1 1 0 1									1	2	3	4	…	12	13	14	15														
1 1 1 0										1	2	3	4	…	12	13	14	15													
1 1 1 1											1	2	3	4	…	12	13	14	15												
a_0	1	2	3	4	…	12	13	14	15	16	17	18	19	…	27	28	29	30													
a_{shift}	1	2	3	4	…	12	13	14	15	a_0																					

The abridged operation reduces twice the hardware amount for a shift in comparison with the long shift computing complete $2n$-bit result $a_f=a_{shift}\{1\,,2n\}2^{-2n}$.

4. Floating-point adder checking

We consider a modulo *3* residue checking of the adder in a part of the mantissa processing. The checking of exponents is not considered because of it is executed with fixed-point numbers and does not lead to complexities.

The error detection circuit (EDC) is shown in Fig. 4.

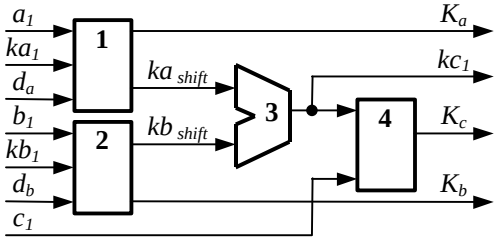

Figure 4: The EDC of floating-point adder

It consists of the blocks 1 and 2 for a check of the shifters, check adder 3 and block 4 for a check of the mantissa result.

The block 1 compares by modulo *3* mantissa a_1 to its input check code ka_1. The result of matching is the error indication code K_a for the mantissa a_1. Besides the block 1 calculates the check code ka_{shift} of the shifted mantissa a_{1shift}. Similarly, the block 2 calculates the code K_b for the mantissa b_1 and code kb_{shift} for the mantissa b_{1shift}.

The block 3 calculates the sum $kc_1=ka_{shift}+kb_{shift}$.

The block 4 compares by modulo *3* mantissa c_1 to its input check code kc_1. The result of matching is the error indication code K_c for the mantissa c_1.

5. Arithmetic shift checking

Traditional design of the block 1 uses the formula (2) received from (1) replacement of numbers by their residuals – check codes:

$$ka_{shift} = ka \times 2^{-d} - ka_0 + ka_s. \qquad (2)$$

The block 1 is shown in Fig. 5 and contains seven units 1÷7.

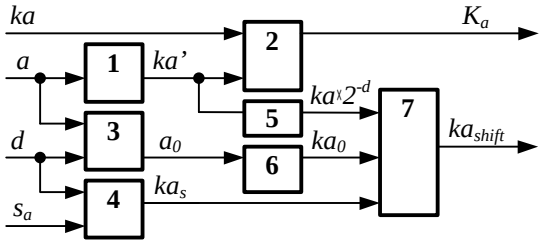

Figure 5: Block 1 of EDC

The unit 1 computes the check code of mantissa $a=a_1$ by the formula $ka'=(a) \bmod 3$. The unit 2 compares calculated ka' and input ka check codes and computes the code K_a. The unit 3 computes abridged bits a_0. The unit 4 calculates the check code ka_s of the sign s_a. The unit 5 computes $ka \times 2^{-d}$. The unit 6 generates $ka_0=(a_0) \bmod 3$. The unit 7 calculates check code ka_{shift} by formula (2).

The unit 3 contains n of *n-to-1* multiplexers with the arithmetic shifter hardware amount $Q_{a.shift}$ and makes together with it long shifter. Thus, the traditional application of a residue checking leads to restoring of the long shift from arithmetic one**.**

The hardware amount for a check is increased on the part with the square dependence on the operand size n and becomes more then the main hardware amount.

Thus, we define a problem of the large hardware amount for on-line testing of the floating-point adder and below we offer the approach to its reduction.

6. Simplification of arithmetic shift checker

The main check hardware amount is located in the unit 3. For the simplification of this unit we transform the formula (1) to formula (3).

$$a_{shift} = (a - a_0 \times 2^d + a_s \times 2^d)\, 2^{-d}. \qquad (3)$$

Then instead of the number a_0 it is necessary to compute the number $a_{01}=a_0 \times 2^d$. The conversion a_0 in a_{01} is shown in table 2.

The number a_{01} is calculated on multiplexers with the functions f_i described in columns $i=1,n$. The part of functions depends only on a part of the code d. The reduction of the multiplexer address size on one bit simplifies the multiplexer more, than in 2 times.

Table 2: Conversion a_0 in a_{01}

d	f_i, $i=1,n$															
4 3 2 1	1	2	3	4	…	12	13	14	15							
2^3 2^2 2^1 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…	2^{12}	2^{13}	2^{14}	2^{15}							
0 0 0 0																
0 0 0 1									15	15						
0 0 1 0								14	15	14	15					
0 0 1 1							13	14	15	13	14	15				
0 1 0 0						12	13	14	15	12	13	14	15			
…					…					…						
1 1 0 0				4	…	12	13	14	15	4	…	12	13	14	15	
1 1 0 1			3	4	…	12	13	14	15	3	4	…	12	13	14	15
1 1 1 0		2	3	4	…	12	13	14	15	2	3	4	…	12	13	14	15
1 1 1 1	1	2	3	4	…	12	13	14	15	1	2	3	4	…	12	13	14	15
	$a_{01}=a_0 \times 2^d$									a_0						

The next theorem defines the summarized simplification of the unit 3.

Theorem 1. The function f_i does not depend from z low bits of the code d, where $i=n+1-2^z(2x_z-1)$, $x_z=1,(n+1)/2^r$, $z=0,r-1$

Proof. The function $f_i=0$ for $d=0,n-i$ and $f_i=i$ for $d=n+1-i,n$. That the function f_i did not depend from z low bits of the code d these bits should be equal to zero for $d=n+1-i$. Substituting value i we obtain $d=2^z(2x_z-1)$, that is $(d) \bmod 2^z=0$. ***Q. E. D.***

The theorem defines, that half of all functions f_i depends on all r bits of the code d and is calculated on *n-to-1* multiplexers with the hardware amount q. The quarter of all functions f_i depends on *r-1* high bits of the code d and is calculated on *n/2-to-1* multiplexers with the hardware amount $q/2$ e. c. For $n=15$ eight functions f_i with an odd i depend on four bits $d\{1,4\}$, four functions f_2, f_6, f_{10}, f_{14} depend on three bits $d\{2,4\}$, two functions f_4, f_8 depend on *2* bits $d\{3,4\}$, one function $f_8=a\{8\} \dot{\vee} d\{4\}$ depends on a bit $d\{4\}$.

The hardware amount Q_{01} for the computation of the number a_{01} is defined as half of all multiplexers by complexity q, quarter of all multiplexers by complexity $q/2$ e. c., that is $Q_{01}=n/2 \times q+ n/4 \times q/2+...+ n/2^{r-1} \times q/2^r=2nq/3$.

Thus, the hardware amount of the unit 3 is reduced in $s_1=1.5$ times.

The formula (3) is transformed in (4):

$$ka_{shift} = (ka - ka_{01} + ka_{s1})\, 2^{-d\{1\}}, \qquad (4)$$

where $ka_{01} = (a_{01}) \bmod 3$, $ka_{s1}=(a_s \times 2^d) \bmod 3$.

The next simplification of the unit 3 is based on the transition from the bit weights 2^{-i} of the binary number to the bit weights 1 or 2 of its check code: $(2^{-i}) \bmod 3=1$ for odd i and $(2^{-i}) \bmod 3=2$ for even i.

In the table 3 the code a_{02} is received by the replacement of the bits the number a_{01}. The bit i of the number a_{01} allocates in column j and row d, if $d\{u\}=1$, where

$$i=n+1-2^u(d/2^u+1)+j, \qquad u=\log_2 j+1, j=1,2^r-1.$$

The column j describes the function F_j for a bit of the code a_{02}. If $d\{u\}=0$, then $F_j=0$.

Table 3: Bit functions of the code a_{02}

d				F_j, $j=1,2^r$														
4	3	2	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
				1	2	1	2	1	2	1	2	1	2	1	2	1	2	1
0	0	0	0															
0	0	0	1	15														
0	0	1	0		14	15												
0	0	1	1	13	14	15												
0	1	0	0				12	13	14	15								
0	1	0	1	11			12	13	14	15								
0	1	1	0		10	11	12	13	14	15								
0	1	1	1	9	10	11	12	13	14	15								
1	0	0	0								8	9	10	11	12	13	14	15
1	0	0	1	7							8	9	10	11	12	13	14	15
1	0	1	0		6	7					8	9	10	11	12	13	14	15
1	0	1	1	5	6	7					8	9	10	11	12	13	14	15
1	1	0	0				4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	3			4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	0		2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				a_{02}														

Theorem 2. The numbers a_{01} and a_{02} have equal check codes for odd n and inverse check codes for even n that is $ka_{01}=(-1)^{n+1}ka_{02}$.

Proof. We consider the difference $i-j=n+1-2^u(d/2^u+1)$, received from (5). It is an even number for odd n. This implies, that bit i of the number a_{01} moves from column i to column j of the code a_{02} without change of its modulo 3 weight, that is $(2^{-i})\mathrm{mod}3=(2^{-j})\mathrm{mod}3$ and $ka_{01}=ka_{02}$. For even n the difference $i-j$ is an odd number and modulo 3 weight of the bit changes to inverse, that is $(2^{-i})\mathrm{mod}3=-(2^{-j})\mathrm{mod}3$ and $ka_{01}=-ka_{02}$. ***Q. E. D.***

From the theorem the conversion (4) in (6) follows.

$$ka_{shift} = (ka -(-1)^{n+1}ka_{02} + ka_s \cdot 2^{d\{1\}})\, 2^{-d\{1\}}, \qquad (6)$$

where $ka_{02} = (a_{02})\mathrm{mod}3$.

Theorem 3. The function F_j does not depend from $u-1$ low bits of the code d.

Proof. The function $F_j=0$ for $d\{u\}=0$ and $F_j=i$ for $d\{u\}=1$. Therefore, at change of bits $d\{u-1\}$, $d\{u-2\}$, …, $d\{1\}$ the functions F_j accept constant values 0 or i and consequently does not depend on bits $d\{1,u-1\}$. ***Q. E. D.***

From the theorem follows, that one function F_1 depends on all r of bits of the code d, two functions F_2, F_3 depend on $r-1$ high bits of the code d e. c. For $n=15$ the function F_1 depends on four bits $d\{1,4\}$, 2 functions F_2, F_3 depend on three bits $d\{2,4\}$, four functions F_4,F_7 depend on two bits $d\{3,4\}$, eight functions F_4,F_7 depend on one bit $d\{4\}$.

The functions F_j are calculated on 2^{r-u}-*to-1* multiplexers with enable. The enable input is connected with the bit $d\{u\}$. Thus, there are used a 2^{r-u-1}-*to-1* multiplexer with the hardware amount $q/2$, two 2^{r-u-2}-*to-1* multiplexers with the hardware amount $q/4$ e. c. The total hardware amount for computation of all the functions F_j calculates as $Q_{02}=1\cdot q/2+2\cdot q/4+...+2^{r-1}\cdot q/2^r=rq/2$.

The unit 3 is simplified in $s_{02}=2n/r$ times. For $n=15$ $s_{02}=7.5$.

The next step on simplification of the EDC is based on application of the check codes ka_V of the operand parts a_V as has been shown for abridged multiplication in [3]. These codes are calculated in process of computation the code ka in the unit 3 by the next formulas:

$ka_{4,7}\{1, 2\}=(a\{4,7\})\mathrm{mod}3$;

$ka_{12,15}\{1, 2\}=(a\{12,15\})\mathrm{mod}3$;

$ka_{8,15}\{1,2\}=(a\{8,11\}+ka_{12,15}\{1,2\})\mathrm{mod}3$;

$ka_{1,15}\{1,2\}=(a\{1,3\}+ka_{4,7}\{1,2\}+ka_{8,15}\{1,2\})\mathrm{mod}3$,

where $a\{4,7\}$, $a\{12,15\}$, $a\{8,15\}$ – the parts a_V of the code $ka=ka_{1,15}\{1,2\}$.

We transform table 4 in table 5 by the change the parts a_V for their check codes ka_V. This transformation defines the code a_{03} with check code $ka_{03}=ka_{02}$, where $ka_{03}=(a_{03})\mathrm{mod}3$. The bits of the code $a_{03}\{1,2r-1\}$ are calculated by functions V_l, $l=1,2r-1$ (table 5). The amount of functions V_l is reduced in $n/(2r-1)$ times. The hardware amount for calculation of the code a_{03} is equal $Q_{03}=1\cdot q/2+2\cdot q/4+...+2\cdot q/2^r=3q/2$.

Thus, the unit 3 is simplified in $s_{03}=2n/3$ times. For $n=15$ $s_{03}=10$.

Table 5: Bit functions V_l of code a_{03}

d				V_l, $l=1,2r-1$						
4	3	2	1	1	2	3	4	5	6	7
2^3	2^2	2^1	2^0	1	2	1	2	1	2	1
0	0	0	0							
0	0	0	1	15						
0	0	1	0		14	15				
0	0	1	1	13	14	15				
0	1	0	0				$ka_{12,15}\{2\}$	$ka_{12,15}\{1\}$		
0	1	0	1	11			$ka_{12,15}\{2\}$	$ka_{12,15}\{1\}$		
0	1	1	0		10	11	$ka_{12,15}\{2\}$	$ka_{12,15}\{1\}$		
0	1	1	1	9	10	11	$ka_{12,15}\{2\}$	$ka_{12,15}\{1\}$		
1	0	0	0						$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	0	0	1	7					$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	0	1	0		6	7			$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	0	1	1	5	6	7			$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	1	0	0				$ka_{4,7}\{2\}$	$ka_{4,7}\{1\}$	$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	1	0	1	3			$ka_{4,7}\{2\}$	$ka_{4,7}\{1\}$	$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	1	1	0		2	3	$ka_{4,7}\{2\}$	$ka_{4,7}\{1\}$	$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
1	1	1	1	1	2	3	$ka_{4,7}\{2\}$	$ka_{4,7}\{1\}$	$ka_{8,15}\{2\}$	$ka_{8,15}\{1\}$
							ka_{03}			

The simplified block 1 is shown in Fig. 6.

It consists of the same units 1÷7 as in the block shown in Fig. 5, but with such differences:

- The unit 1 calculates also codes ka_V without additional hardware amount and delay.
- The unit 3 is simplified multiply.
- The unit 6 is simplified in $n/(2r-1)$ times as size of its input code is reduced from n up to $(2r-1)$ bits.
- The unit 1 works concurrently with the block for the exponent processing before calculation of the code d, that essentially raises speed of the EDC.

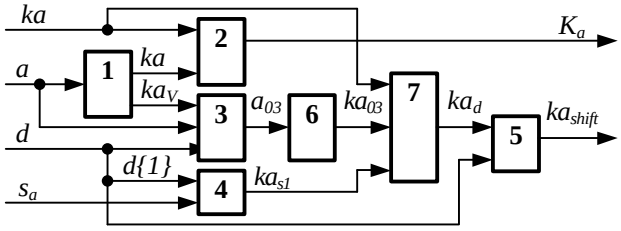

Figure 6: Simplified block 1 of EDC

The units 1 and 2 designed on modulo *3* adders *1¸8* with fast pyramid structure are shown in Fig. 7.

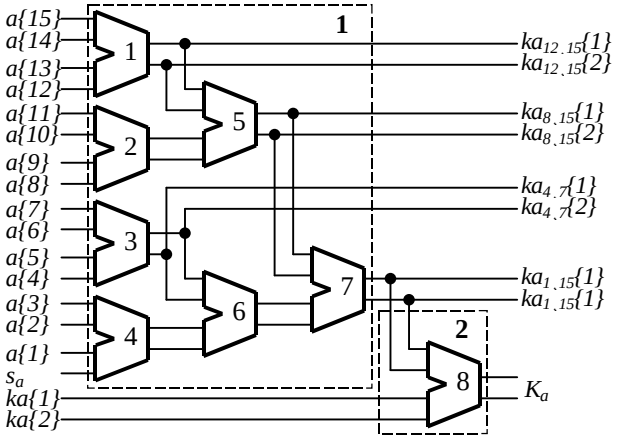

Figure 7: Units 1 and 2 of the block 1

The units 3 and 4 are shown in Fig. 8. The unit 3 calculates the code ka_{03} on multiplexers *1¸7* with enable. The unit 4 calculates the code $ka_s=s_a$Ùd{1} on gate *8* AND.

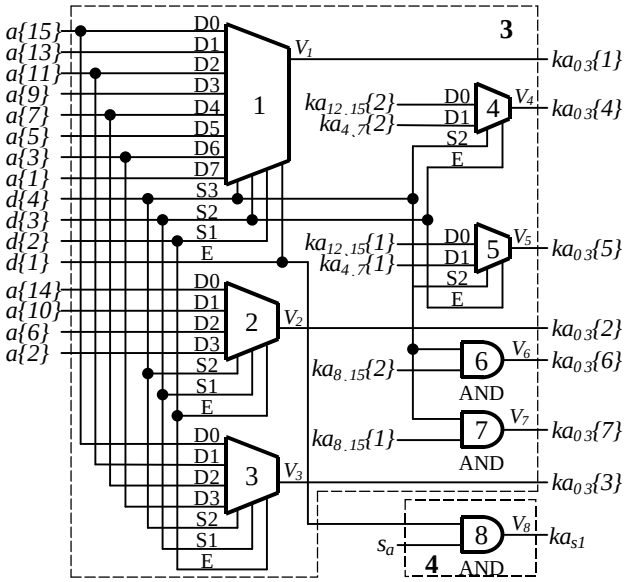

Figure 8: Units 3 and 4 of the block 1

The unit 5 calculates the code ka_{shift}*{1,2}=*ka_d{1,2}Åd{1}* on two gates XOR.

The units 6 and 7 calculate the code $ka_d=ka-ka_{03}+ka_{s1}$ and are designed on modulo 3 adders as a uniform circuit with pyramidal structure similarly to a unit 1.

7. Result

We have designed the floating-point adder and its EDC on FPGA Xilinx ÕÑ4000 for *n=15*. The floating-point adder, the EDC with the traditional and offered structure are executed on 151, 183 and 41 Configurable Logic Blocks. Thus, the hardware amount for a check is reduced in 4.5 times (from 183/151=121% to 41/151=27%). The delay for computation the check code ka_{shift} is reduced in 1.9 times (from 28.9 ns to 15.0 ns). The simplification of the EDC has been achieved without reduction of detection capability.

8. Conclusion

We have presented a new residue checking method for on-line testing of floating-point adder. In the method the problem of the large hardware amount for a check of the abridged operation (arithmetic shift) executed in floating-point adder has been solved. This hardware amount was reduced from square dependence on the operand size to linear one. The significant simplification of error detection circuit was valid by its design on FPGA Xilinx. The solutions based on this method have been found industrial application in digital signal processing system for fast Fourier transform execution [4, 5].

9. References

[1] Sparmann U., Reddy S., “On the Effectiveness of Residue Code Checking For Parallel Two’s Complement Multipliers”, Proc. 24th Fault Tolerant Computing Symposium, Austin Texas, June 1994.

[2] Drozd A. V., Lobachev M. V., Hassonah W. “Hardware Check of Arithmetic Devices with Abridged Execution of Operations”, Proc. the European Design & Test Conference (ED & TC 96), Paris, France, March 11 – 14, 1996. – p. 611.

[3] Drozd A.V., “Control in Modulus of the Single-Cycle Multiplier with Abridged Mode of Operation”, Engineering Simulation. Vol. 16. – 1999. – pp. 377 – 385.

[4] Drozd A.V., Polin E. L., at al., “Circuit for information shift”, SU 1277116, G 06 F 11/10, 1986.

[5] Drozd A.V., Polin E. L., at al., “Circuit for floating-point operand addition with checking”, SU 1310826, G 06 F 11/10, 1987.