Improvement of Standard and Non-Standard Floating-Point Operators

Pongyupinpanich Surapong[1], **François Philipp**[2], **Faizal Arya Samman**[3], and **Manfred Glesner**[4], Non-members

ABSTRACT

This paper presents the design and analysis of a floating-point arithmetic accelerator in compliance with the IEEE standard single precision floating-point format. The accelerator can be used to extend a general-purpose processor such as Motorola MC6820, where floating-point execution units are unembedded by default. It implements standard and non-standard mathematic functions, addition/subtraction, multiplication, Product-of-Sum and Sum-of-Product through a micro-instruction set supported by both single and multi-processors systems. The architecture of the unit is based on an instruction pipeline which can simultaneously fetch and execute an instruction within one clock cycle. The non-standard operations such as Product-of-Sum and Sum-of-Product are introduced to compute three-input operands. The algorithm complexity and hardware critical delay are determined for each operator. The synthesis results of the accelerator on a Xilinx FPGA Virtex 5 xc5vlx110t-3ff-1136 and on Faraday 130-nm Silicon technology report that the design respectively achieves 200 MHz and 1 GHz.

Keywords: Floating-Point Operators, Accelerator Processor, Product-of-Sum, Sum-of-Product, 32-bit IEEE Standard Single-Precision

1. INTRODUCTION

Requirements for real-time high-accuracy computation considerably increase in recent applications. Critical applications like medical image processing [1] or Linear Phase FIR digital filter [2] rely on floating-point computation for accurate and efficient processing. The majority of modern processors such as Motorola 6840 integrates a hardware floating-point arithmetic unit in order to fulfill these requirements whereas classic processors perform floating-point arithmetic using software libraries. Although the operations can be introduced by this method, the computation is very slow in comparison to hardware implementation.

Several strategies for the implementation of floating-point accelerators were reported in related works. The first projects focused on chip design and functionality. In 1983, Huntsman et al. [3] introduced the MC68881 floating-point co-processor used to cooperate with Motorola's M68000 32-bit processors family. The MIPS R3010 chip [4] specified for the R3000 RISC processor was proposed in order to reduce design costs. It provides the basic floating-point operations, Addition/Subtraction, Multiplication, and Division. Maurer [5] introduced the WE32106 math accelerator, but mainly focused on verification techniques. Nakayama et al. [6] designed a 80-bit floating-point co-processor providing 24 instructions and 22 mathematic functions where Adder/Subtractor and Multiplier were designed in pipelining structure, but Divider was performed using the CORDIC algorithm. Kawasaki et al. [7] introduced a pipelined floating-point co-processor cooperating with the GMICROs processor as an intelligent CPU for TRON architecture. The co-processor has 23 instructions to perform basic and trigonometric operations.

Secondly, the improvement of performance and efficiency at runtime was investigated. Darley et al. [8] proposed the TMS390C602A floating-point co-processor to cooperate with the SPARC TMS390C601 integer processor. They optimized the system performance by balancing the floating-point execution throughput and instruction fetching. This method demonstrated higher performance while dramatically cutting system costs. A 16-bit pipelining floating-point co-processor on FPGA was investigated by Fritz and Valerij in [9]. Based on the SIMD structure, the co-processor is placed in between a processor and the main memory. When the processor needs to execute a floating-point operation, the processor will simultaneously send an instruction to the co-processor and the address of the given operands to the memory. The co-processor can thus directly fetch the operands from the memory.

The enhancement of designs and algorithms of basic arithmetic units was the third strategy. Nielsen et al. [10] proposed a pipelined floating-point addition algorithm with 4-state in packet forwarding format, which was a redundant representa-

Manuscript received on August 1, 2011 ; revised on January 2, 2012.

[1,2,3,4]The authors are with Microelectronic Systems Research Group, Faculty of Electrical Engineering and Information Technology, Technische Universität Darmstadt, Merckstrasse 25 64283 Darmstadt Germany., E-mail: surapong@mes.tu-darmstadt.de, francois.philipp@mes.tu-darmstadt.de, faizal.samman@mes.tu-darmstadt.de and glesner@mes.tu-darmstadt.de

tion of the floating-point number, in order to improve the mantissa fraction. Chen et al. [11] introduced the architecture of a multiplication-add fused (MAF) unit to reduce the three-word-length addition to two-word-length for carry propagation in conventional MAF. Either Leading-One/Zero-detection or-prediction, common functions for floating-point operations, were considered by Javier et al. [12], Suzuki et al. [13], Hokenek et al. [14], and Schmookler et al. [15].

In hard real-time computation such as digital filter application [16], time constraint is a main factor for design consideration, where calculation has to be finished before a new sample arrives. If the floating-point computation units are performed by using software library on a process, which obviously provides longer latency than hardware, the targeting time constraint cannot be achieved. Clearly, modern processors where the floating-point units are embedded can fulfill the requirement. In floating-point units, critical delay comes from Leading-One-Detection, Shift functions and integer multiplier. To reduce this delay, the common functions have to be investigated and improved. Multi-processor system can accelerate an application's computation. Normally, the processors execute their floating-point tasks by their own floating-point library which consume more resources and time. Thus, hardware-sharing concept where one floating-point accelerator is shared for multi-processor will not only reduce the consumed resources, but also computation time and power consumption.

2. CONTRIBUTION

In accordance with the three aforementioned strategies, we propose the design of a novel pipelined floating-point accelerator to supporting the following requirements :

- **Architecture**: we design a floating-point accelerator providing high performance. It has to minimize the redesign costs to cooperate with general purpose processors which do not integrate floating-point arithmetic units.
- **Performance**: we increase and balance the performance and efficiency of the floating-point operators on both standard (2-inputs) and non-standard (3-inputs) when these operators are combined on a single chip. The standard operators are Adder/Subtractor and Multiplier. The mainly used non-standard operators are Product-of-Sum operator and a Sum-of-Product.
- **Extendability**: besides the standard and the non-standard operations, we want to design a floating-point accelerator supporting additional mathematical functions such as trigonometric, linear and hyperbolic functions.

In order to achieve our purposes, we have three main contributions in this paper: 1) analysis and improvement of the performance and the efficiency of the floating-point algorithms on both standard and non-standard operators; 2) introduction of an optimal floating-point unit architecture based on common functions and a partially linear integer multiplier; 3) introduction of a simple micro-instruction set with instruction format and implementation for single-and multiple-processors systems

The rest of the paper is organized as follows. The floating-point algorithms of the standard and non-standard operators are analyzed in Section 3. The design and enhancement of the Leading-One/Zero-Detection and Right/Left shifting functions as well as a partial liner integer multiplier are introduced in Section 4. The implementation and investigation of floating-point operators are considered in Section 5. Section 6 details the design and architecture of floating-point arithmetic accelerator. Finally, Section 7 summarizes and concludes the paper.

3. ANALYSIS OF FLOATING-POINT OPERATION ALGORITHMS

The algorithms of standard operators, i.e., adder/-substractor, multiplier, and non-standard operators, i.e., Product-of-Sum (PoS) operator and Sum-of-Product (SoP) operator, are analyzed and considered to increase computation performance. The IEEE standard single-precision floating-point representation (Fig. 1) is applied in our analysis with $n = 32$, $ne = 8$, and $nf = 23$. In order to reduce the design complexity, rounding algorithms used to approximate an intermediate mantissa fraction are ignored.

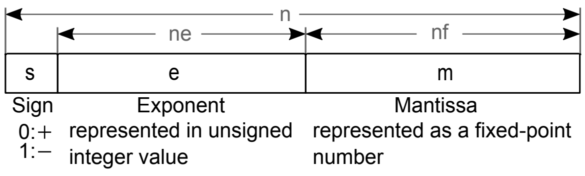

n
ne
nf
s
e
m
Sign
0:+
1:−
Exponent
represented in unsigned integer value
Mantissa
represented as a fixed-point number

Fig.1: *IEEE standard single-precision representation*

3.1 Common Functions

The Unpacking, Comparison, and Norm functions, which are commonly used to perform the floating-point operation algorithms are discussed in the following.

3.1.1 Function Unpacking

This function shown in Alg. 1 extracts the two input operands into two groups A and B of Sign, Exponent and Mantissa fraction: A$_s$, A$_e$, and A$_m$ for group A and B$_s$, B$_e$, and B$_m$ for group B. The carry-bit and guard-bit, b'01, are padded on the MSB of the mantissa fraction for computation, where || denotes concatenation operation.

For example, we assumed that op$_1$ $= -100(h'C2C$ 80000), op$_2$ $=$ $200(h'43480000)$, n = 32, ne = 8, nf = 23. After executing the unpacking function, its output results will be $A_s = b'1$, $A_e = b'10000101$,

Alg. 1 Unpacking(op$_1$,op$_2$,n,ne,nf)

```
1: A_s=op1(n-1), A_e=op1(n-2:n-ne-2);
2: A_m=b'01||op1(nf-1:0);
3: B_s=op2(n-1), B_e=op2(n-2:n-ne-2);
4: B_m=b'01||op2(nf-1:0);
5: return A_s, A_e, A_m, B_s, B_e, B_m
```

A_m=b'01100100000000000000000000, B_s=b'0, B_e= b'10000110, B_m=b'01100100000000000000000000.

3.1.2 Function Comparison

The function compares the two input operands fractioned into group A and B. The comparison is normally done using an If-Statement, where the two signs, A$_s$ and B$_s$, are first compared, followed by a comparison of the two exponents A$_e$ B$_e$ and mantissas A$_m$ B$_m$. By means of this method, a critical delay appears. In order to minimize the critical delay, a parallel comparison based on combinational circuit is introduced. The truth-Table 1 shows possible cases of the operand A and B, where p, q, and z depend on A$_e$, B$_e$, A$_m$, and B$_m$.

Table 1: *Representing the relationship between operand A and B in keeping with g$_e$ and g$_m$ in truth-table.*

case	g_e	g_m	$g_{A>B}$	$g_{A=B}$
1	p	p	1	0
2	p	q	1	0
3	p	z	1	0
4	q	p	1	0
5	q	q	0	1
6	q	z	0	0
7	z	p	0	0
8	z	q	0	0
9	z	z	0	0

For instance, assume that g$_e$ and g$_m$ are greatening results of exponent and mantissa values of two input operands. We define that p=b'01, q=b'11, and z=b'00. The 2nd case where g$_e$=p and g$_m$=q means that A$_e$ is greater than B$_e$ and A$_m$ is the same as B$_m$. The two outputs g$_{A>B}$ and g$_{A=B}$ are respectively set to b'1 and b'0 where they cover the condition that the operand A is greater than the operand B. The 5th case shows the condition that if the operand A is equal to the operand B, then g$_e$=q and g$_m$=q, where the two outputs are set to b'0 an b'1. Alg. 2 presents the comparison function derived from Table 1.

Form the previous computational results where A_s=b'1, A_e=b'10000101, A_m=b'011001000000000000 0000000, B_s=b'0, B_e=b'10000110, B_m=b'011001000 0000 000000000000, after the function Comparison is executed, g_e=b'00 and g_m=b'11. The output results $g_{A=B}$ and $g_{A>B}$ will equal to $b'0$ and $b'0$, which is the 8th case on Table 1.

Alg. 2 Comparison(A_s,B_s,A_e,B_e,A_m,B_m)

```
1: if (A_e >B_e)s then
2:     g_e=b'01;
3: else if A_e =B_e then
4:     g_e=b'11;
5: else
6:     g_e=b'00;
7: end if
8: if (A_m >B_m) then
9:     g_m=b'01;
10: else if (A_m =B_m) then
11:     g_m=b'11;
12: else
13:     g_m=b'00;
14: end if
15: g_{A>B}=((~g_e(1))·g_e(0))·((~g_m(1))+g_m(0)))+(g_e(0) · (~g_m(1))·g_m(0));
16: g_{A=B}=g_e(1)·g_e(0)·g_m(1)·g_m(0);
17: return g_{A>B}, g_{A=B}
```

3.1.3 Function Norm

The sign (Sign), exponent (E), and mantissa (M) are normalized in accordance with the IEEE standard single-precision format. The mantissa M is shifted to the MSB depending on the given Position parameter. Simultaneously, the exponent E is either added or subtracted, according to the carry-bit and guard-bit of the mantissa M. The sign and the adapted exponent and mantissa are finally packed together. The Norm function is illustrated in Alg. 3.

Alg. 3 Norm(Sign,E,M,Position, n, ne, nf)

```
1: X(n-1)=Sign;
2: if (M(nf+1:nf)=b'10)or(M(nf+1:nf)=b'11) then
3:     X(n-2:n-ne-1)=E+1
4:     X(nf-1:0)=M(nf:1)
5: else
6:     X(n-2:n-ne-1)=E-Position
7:     X(nf-1:0)=Shift(M,Position,Left)
8: end if
9: return X
```

3.1.4 Function Unpacking3

This is a common function for non-standard operators. It is similar to the Unpacking function, but there are 3-input operands, op$_1$, op$_2$, and op$_3$ as shown in Alg. 4. The output are split into three groups of Sign, Exponent, and Mantissa fractions, A$_s$, A$_e$, A$_m$, B$_s$, B$_e$, B$_m$, C$_s$, C$_e$, and C$_m$ respectively.

Alg. 4 Unpacking3(op$_1$,op$_2$,op$_3$,n,ne,nf)

```
1: A_s=op1(n-1), A_e=op1(n-2:n-ne-2);
2: A_m=b'01||op1(nf-1:0);
3: B_s=op2(n-1), B_e=op2(n-2:n-ne-2);
4: B_m=b'01||op2(nf-1:0);
5: C_s=op3(n-1), C_e=op3(n-2:n-ne-2);
6: C_m=b'01||op3(nf-1:0);
7: return A_s, A_e, A_m, B_s, B_e, B_m, C_s, C_e, C_m
```

3.2 Standard Operation

3.2.1 Floating-Point Addition/Subtraction

The floating-point addition/subtraction algorithm, detailed by Alg. 5, is compounded of the common functions which are introduced in section 3.1 The algorithm is built in respect of design simplicity and implementation in digital hardware.

Alg. 5 Floating-Point Adder/Subtractor

```
Require: op_1, op_2, n, ne, nf, sub
    {Step 1: unpacking}
1:  $op_2(n-1)$=sub $\oplus$ $op_2(n-1)$
2:  [$A_s$,$A_e$,$A_m$,$B_s$,$B_e$,$B_m$]=Unpacking($op_1$,$op_2$,n,ne,nf)
    {Step 2: comparing and Sing evaluation}
3:  [$g_{A>B}$, $g_{A=B}$]=Comparison($A_s$,$B_s$,$A_e$,$B_e$,$A_m$,$B_m$)
4:  Sign=($A_s \cdot B_s$)+($A_s \cdot g_{A>B}$ $\cdot$ ($\sim g_{A=B}$))+($B_s$ $\cdot$ ($\sim g_{A>B}$) $\cdot$
    ($\sim g_{A=B}$))
    {Step 3: Exponent subtraction, mantissa swap, and shift}
5:  if ($g_{A>B}$ =b'1) or ($g_{A=B}$ =b'1) then
6:      $E_{add}$=$A_e$, $M_{l1}$ =$A_m$, $M_{l2}$ =$B_m$
7:      $Shift_{length}$ =$A_e$−$B_e$
8:  else
9:      $E_{add}$=$B_e$, $M_{l1}$ =$B_m$, $M_{l2}$ =$A_m$
10:     $Shift_{length}$ =$B_e$−$A_e$
11: end if
12: $M_{adp}$=Shift($M_{l2}$,$Shift_{length}$,Right)
    {Step 4: Mantissa addition/subtraction}
13: if (($A_s \oplus B_s$)=b'0) then
14:     $M_{add}$=$M_{l1}$+$M_{adp}$
15: else
16:     $M_{add}$=$M_{l1}$-$M_{adp}$
17: end if
    {Step 5: Leading-One-Detection and normalization}
18: Position=LOD($M_{add}$)
19: X=Norm(Sign,$E_{add}$,$M_{add}$,Position, n, ne, nf)
```

The proposed algorithm has been split in five steps for both addition and substraction as described by the following points:

• **Step 1** Unpacking: the sign bits of the two operands, op_1 and op_2, are first evaluated by a XOR operation with the *sub* parameter. Afterwards, both operands will be fractioned into 3 main triples, Sign, Exponent, and Mantissa, by the Unpacking function, which outputs the parameters A_s, A_e, A_m, B_s, B_e, and B_m respectively.

• **Step 2** Comparing and Sign evaluation: the partial exponents and mantissas, A_e, A_m, B_e, and B_m, are compared using the Comparison function to determine the greatest value between op_1 and op_2. Then, the Sign is evaluated by the optimized combinational logic.

• **Step 3** Exponent subtraction and mantissa swap: the results of the comparison $g_{A>B}$ and $g_{A=B}$ are used to compute the difference of the two exponents $Shift_{length}$ and to swap the exponents and the mantissas. The $Shift_{length}$ will be used to adjust the lower mantissa M_{l2} using the Shift function.

• **Step 4** Mantissa Addition/Subtraction: the addition/subtraction of the mantissas depends on the xoring result of A_s and B_s

• **Step 5** Leading-One-Detection and Normalization: the first bit one of the addition/subtraction result of the mantissas is searched by the LOD function, where the detected result will be used to normalize the final exponent and the final mantissa generated from previous steps. The Norm function will finally pack them together.
The details of the LOD and Shift functions have been fully described in section 4.

3.2.2 Floating-Point Multiplication

In comparison with the floating-point addition/subtraction algorithm, the complexity of the floating-point multiplication algorithm is lower as illustrated in Alg. 6. It is also performed in five steps described as follows:

• **Step 1** Unpacking: the two operands are fractioned by the Unpacking function.

• **Step 2** Subtracting and comparing: the Comparison function is applied to compare the exponents and the mantissa, A_e, B_e, A_m, and B_m.

• **Step 3** Swap and Sign evaluation: the variable Swap and Sign are evaluated using AND and XOR operations. The Swap variable is then used to perform swapping of the two exponents and the two mantissas.

• **Step 4** Exponent and mantissa determination: the two exponents, E_{l1} and E_{l2}, are subtracted and added by 127 in the signed integer form in order to output the final exponent E_{mul}. The unsigned integer multiplication is utilized to compute the final mantissa M_{mul}.

• **Step 5** Leading-One-Detection and Normalization: The process is the same as the Step 5 of the addition/-subtraction algorithm.

Alg. 6 Floating-Point Multiplier

```
Require: op_1, op_2, n, ne, nf
    {Step 1: unpacking}
1:  [$A_s$,$A_e$,$A_m$,$B_s$,$B_e$,$B_m$]=Unpacking($op_1$,$op_2$,n,ne,nf)
    {Step 2: subtracting and comparing}
2:  [$g_{A>B}$, $g_{A=B}$]=Comparison($A_e$,$B_e$,$A_m$,$B_m$)
    {Step 3: Swap and Sign evaluation}
3:  Swap=($\sim g_{A>B}$)$\cdot$($\sim g_{A=B}$)
4:  Sign=$S_A \oplus S_B$
5:  if (Swap=b'0) then
6:      $E_{l1}$=$A_e$, $E_{l2}$=$B_e$, $M_{l1}$ =$A_m$, $M_{l2}$ =$B_m$
7:  else
8:      $E_{l1}$=$B_e$, $E_{l2}$=$A_e$, $M_{l1}$ =$B_m$, $M_{l2}$ =$A_m$
9:  end if
    {Step 4: Exponent and Mantissa determination}
10: $E_{mul}$=$E_{l1}$-$E_{l2}$+127
11: $M_{mul}$=$M_{l1}$$\times$ $M_{l2}$
    {Step 5: Leading-One-Detection and normalization}
12: Position=LOD($M_{mul}$)
13: X=Norm(Sign,$E_{mul}$,$M_{mul}$,Position, n, ne, nf)
```

3.3 Non-Standard Operation

There are non-standard arithmetic operations with 3-input operands which are being widely used in digital signal processing applications. Product-of-Sum operator (PoS) and Sum-of-Product operator (SoP), (A+B)× C and (A×B)+C, are frequently employed in multimedia and filtering applications [17] and [16]. These operators can be performed using basic addition and multiplication in cascade. However, in order to improve the performance and the efficiency of the floating-point unit, algorithms for the PoS and SoP operators are introduced by the fusion of the floating-point addition and multiplication algorithms.

3.3.1 Floating-Point Product-of-Sum Operation

The floating-point PoS operator, (A+B)× C, is a combination of the floating-point adder and multiplier. The PoS algorithm shown in Alg. 7 is described by the following points:

Alg. 7 Floating-Point Product-of-Sum(PoS)

```
Require: $op_1$, $op_2$, $op_3$, n, ne, nf
    {Step 1: Unpacking}
1:  [$A_s$,$A_e$,$A_m$,$B_s$,$B_e$,$B_m$,$C_s$,$C_e$,$C_m$]=
    Unpacking3($op_1$,$op_2$,$op_3$,n,ne,nf)
    {Step 2: Comparing and Sign evaluation}
2:  [$g_{A>B}$, $g_{A=B}$]=Comparison($A_e$,$B_e$,$A_m$,$B_m$)
    {Step 3: Exponent subtraction, Mantissa swap, and final
    Sign evaluation}
3:  $\mathrm{Sign}_{pos}$=$\mathrm{Sign}_l \oplus \mathrm{C}_s$
4:  $\mathrm{Sub}_{l1}$=$\mathrm{A}_s \oplus \mathrm{B}_s$
5:  if ($\mathrm{g}_{A>B}$=b'1) or ($\mathrm{g}_{A=B}$=b'1) then
6:      $\mathrm{E}_{l1}$=$\mathrm{A}_e$-$\mathrm{B}_e$, $\mathrm{E}_{l2}$=$\mathrm{B}_e$, $\mathrm{M}_{l1}$=$\mathrm{A}_m$, $\mathrm{M}_{l2}$=$\mathrm{B}_m$
7:  else
8:      $\mathrm{E}_{l1}$=$\mathrm{B}_e$-$\mathrm{A}_e$, $\mathrm{E}_{l2}$=$\mathrm{A}_e$, $\mathrm{M}_{l1}$=$\mathrm{B}_m$, $\mathrm{M}_{l2}$=$\mathrm{A}_m$
9:  end if
    {Step 4: Exponent and Mantissa determination}
10: if ($\mathrm{g}_{A>B}$=b'1) then
11:     $\mathrm{E}_{add}$=$\mathrm{E}_{l2}$-$\mathrm{C}_e$+127
12: else
13:     $\mathrm{E}_{add}$=$\mathrm{C}_e$-$\mathrm{E}_{l2}$+127
14: end if
15: $\mathrm{M}_{shift}$=b'0||Shift($\mathrm{M}_{l2}$,$\mathrm{E}_{l1}$,Right)
16: if ($\mathrm{Sub}_{l1}$=b'0) then
17:     $\mathrm{M}_{add}$=$\mathrm{M}_{l1}$+$\mathrm{M}_{shift}$
18: else
19:     $\mathrm{M}_{add}$=$\mathrm{M}_{l1}$-$\mathrm{M}_{shift}$
20: end if
    {Step 5: Leading-One-Detection, final Exponent and
    Mantissa alignment}
21: Position=LOD($\mathrm{M}_{add}$)
22: if ($\mathrm{M}_{add}$(nf+1:nf)=b'10) or ($\mathrm{M}_{add}$(nf+1:nf)=b'11) then
23:     $\mathrm{E}_{pos}$=$\mathrm{E}_{l1}$+$\mathrm{E}_{add}$+1
24:     $\mathrm{M}_{align}$=b'01||$\mathrm{M}_{add}$
25: else
26:     $\mathrm{E}_{pos}$=$\mathrm{E}_{l1}$+$\mathrm{E}_{add}$-Position
27:     $\mathrm{M}_{align}$=b'01||Shift($\mathrm{M}_{add}$,Position,Left)
28: end if
    {Step 6: final Mantissa determination}
29: $\mathrm{M}_{pos}$=$\mathrm{M}_{align} \times \mathrm{C}_m$
    {Step 7: Leading-One-Detection and normalization}
30: Position=LOD($\mathrm{M}_{pos}$)
31: X=Norm($\mathrm{Sign}_{pos}$,$\mathrm{E}_{pos}$,$\mathrm{M}_{pos}$,Position, n, ne, nf)
```

• **Step 1** Unpacking: the three operands, op_1, op_2, op_3, are fractioned in A_s, A_e, A_m, B_s, B_e, B_m, C_s, C_e, and C_m by by the Unpacking3 function.

• **Step 2** Comparing and Sign evaluation: the two exponents, A_e and B_e, and the two mantissas, A_m and B_m, are sorted by th function Comparison. Then, the sign is determined.

• **Step 3** Exponent subtraction, Mantissa swap, and final Sign evaluation: the final sign Sign_{pos} is evaluated by XORing Sign_l and C_s. Meanwhile, the sign bit Sub_l is XORed by A_s and B_s. The comparison results will be used to swap mantissas and to compute the exponent difference.

• **Step 4** Exponent and Mantissa determination: the exponents and mantissas are computed by adding and shifting.

• **Step 5** Leading-One-Detection, final Exponent, and Mantissa alignment: the first bit is searched by the LDO function. The final exponent and mantissa alignment corresponding to the addition result of the mantissa of op_1 and op_2 are accumulated by adding and shifting.

• **Step 6** Final mantissa determination: the two mantissas are multiplied using unsigned integer multiplication.

• **Step 7** Leading-One-Detection and normalization : operating as the step 5 of the Alg. 5

3.3.2 Floating-Point Sum-of-Product Operation

The floating-point Sum-of-Product (SoP) operation algorithm ((A×B)+C) is detailed by the following:

• **Step 1** Unpacking : operating as the step 1 of the Alg. 7.

• **Step 2** Comparing, Exponent subtraction, Mantissa swap, and Sign evaluation: the two exponents, A_e and B_e, and the two mantissas, A_m and B_m, are compared by function Comparison. The comparison result will be used to swap the mantissas, to compute an intermediate exponent, and to evaluate the sign by xoring A_s and B_s.

• **Step 3** Mantissa multiplication: the two mantissas are multiplied in form of unsigned integer multiplication.

• **Step 4** Leading-One-Detection, Exponent, and Mantissa alignment: the length of shifting is determined by function LOD. The exponents are calculated and the mantissas are aligned by addiction and shifting.

• **Step 5** Comparing, final Sign evaluation, Exponent subtraction, and Mantissa swap: the two exponents and the two mantissas are compared and the final sign is determined by XORing. Afterwards, the intermediate mantissas and the intermediate exponents are swapped.

• **Step 6** Shift and final Mantissa determination : the mantissa is aligned and the final mantissa M_{op} is determined.

• **Step 7** Leading-One-Detection and normalization : operating as the step 5 of the Alg. 5

By considering the algorithms of the four floating-point operations described in Alg. 5-8, it can be noticed that common functions and basic mathematical operation used are Right/Left Shifting and Leading-One-Detection (LOD) functions as well as signed integer addition/subtraction and multiplication. These functions and operations have a significant impact on the performance and efficiency of the floating-point units. The critical delays of the LOD and Right/Left

Algorithm 8 Floating-Point Sum-of-Product(SoP)

```
Require: $op_1$, $op_2$, $op_3$, n, ne, nf
     {Step 1: Unpacking}
 1:  [$A_s$,$A_e$,$A_m$,$B_s$,$B_e$,$B_m$,$C_s$,$C_e$,$C_m$]=
     Unpacking3($op_1$,$op_2$,$op_3$,n,ne,nf)
     {Step 2: Comparing, Exponent subtraction, Mantissa
     swap, and Sign evaluation}
 2:  [$g1_{A>B}$, $g1_{A=B}$]=Comparison($A_e$,$B_e$,$A_m$,$B_m$)
 3:  if ($g1_{A>B}$=b'1) or ($g1_{A=B}$=b'1) then
 4:      $M_{l1}$=$A_m$, $M_{l2}$=$B_m$, $E_{l2}$=$B_e$-$A_e$+127
 5:  else
 6:      $M_{l1}$=$B_m$, $M_{l2}$=$A_m$, $E_{l2}$=$B_e$-$A_e$+127
 7:  end if
 8:  $Sign_{mul}$=$A_s \oplus B_s$
     {Step 3: Mantissa multiplication}
 9:  $M_{mul}$=$M_{l1} \times M_{l2}$
     {Step 4: Leading-One-Detection, Exponent and Mantissa
     alignment}
10:  Position=LOD($M_{mul}$)
11:  if ($M_{mul}$(2·nf+1:2·nf)=b'11) or
     ($M_{mul}$(2·nf+1:2·nf)=b'10) then
12:      $E_{mul}$=$E_{l2}$+1, $M_{align}$=b'01||$M_{mul}$
13:  else if ($M_{mul}$(2·nf)=b'1) then
14:      $E_{mul}$=$E_{l2}$, $M_{align}$=$M_{mul}$
15:  else
16:      $E_{mul}$=$E_{l2}$-Position,
         $M_{align}$=b'01||Shift($M_{mul}$,Position)
17:  end if
     {Step 5: Comparing, final Sign evaluation, Exponent sub-
     traction and Mantissa swap}
18:  [$g2_{A>B}$, $g2_{A=B}$]=Compare($E_{mul}$,$C_e$,$M_{align}$,$C_m$)
19:  $Sign_{sop}$=$Sign_{mul} \oplus C_s$
20:  if ($g2_{A>B}$=b'1) or ($g2_{A=B}$=b'1) then
21:      $M_{add_{l1}}$=$M_{align}$, $M_{add_{l2}}$=$C_m$
22:      $E_{sop}$=$E_{mul}$, $E_{add_{l2}}$=$E_{mul}$-$C_e$
23:  else
24:      $M_{add_{l1}}$=$C_m$, $M_{add_{l2}}$=$M_{align}$
25:      $E_{sop}$=$C_e$, $E_{add_{l2}}$=$C_e$-$E_{mul}$
26:  end if
     {Step 6: shift and final Mantissa determination}
27:  $M_{shift}$=b'0||Shift($M_{add_{l2}}$,$E_{add_{l2}}$,Right)
28:  if ($Sign_{sop}$=b'0) then
29:      $M_{sop}$=$M_{add_{l1}}$+$M_{shift}$
30:  else
31:      $M_{sop}$=$M_{add_{l1}}$-$M_{shift}$
32:  end if
     {Step 7: Leading-One-Detection and normalization}
33:  Position=LOD($M_{sop}$)
34:  X=Norm($Sign_{sop}$,$E_{sop}$,$M_{sop}$,Position,n,ne,nf)
```

Shifting functions with For-Loop method and normal shifting method are approximately 10 ns and 6.2 ns at 32-bit data-width. The normal integer multiplier has the critical delay 9.781 ns at 32-bit data-width. Thus, design and analysis of Right/Left Shifting and Leading-One-Detection (LOD) functions as well as the integer multiplication operations are then discussed in Section 4.

4. DESIGN AND ENHANCEMENT OF THE FUNCTION AND OPERATION

4.1 Leading-One-Detection based on Binary-Tree Algorithm

Leading-One-Detection (LOD) is a function used to detect the first bit one from MSB to LSB or vice versa. Normally, LOD is performed by the comparison of two adjacent bits by any Loop statement. By means of this method, the critical delay is proportional to the length of a considered bit string. In order to reduce this critical delay, a binary-tree searching algorithm has been applied instead. The depth of this structure is determined by $log_2(N)$, where N is the size of any input binary string. The Binary-Tree structure is illustrated in Fig. 3.

Table 2 represents the truth table of the Binary-Tree Cell (BT-Cell), where X, C, $N_{v-pq} \in \{0,1\}$ and p, q, $N_{loc-pq} \in \left\{0, .. \frac{N}{2} - 1\right\}$. Assuming that p, q are a coordinate in XY plan, where p is a position on the X axis and q is a position on the Y axis (depth). Thus, BT-Cell01 is Binary-Tree located in the depth level 0 in the column 1. $X_{N-1:0}$ is an input binary vector and C determines the direction of the search (either MSB to LSB or LSB to MSB). N_{loc-pq} is a selected location corresponding to the coordinates p and q. N_{v-pq} is the bit value of the selected location. For instance, in the 6th case, X_N, X_{N-1}, and C are equal to b'101, meaning that the BT-Cell searches the fist bit one from MSB to LSB. Afterwards, N_{v-pq} is set to b'1 and N_{loc-pq} is equal to N.

Table 2: *Binary selection algorithm [18] for the BT-Cell.*

Case	X_N	X_{N-1}	C	Node-loc. (N_{loc-pq})	Node-Value (N_{v-pq})
1	0	0	0	N	0
2	0	0	1	N-1	0
3	0	1	0	N-1	1
4	0	1	1	N-1	1
5	1	0	0	N	1
6	1	0	1	N	1
7	1	1	0	N	1
8	1	1	1	N-1	1

The corresponding optimized combinational logic is described by Eqs. (1) and (2). Fig. 2 illustrates the design and architecture of the Binary-Tree Cell, where $\otimes$ is AND gate and $\oplus$ is OR gate.

$$N_{loc-pq} = \begin{cases} N, & \text{if } X_N \cdot (\overline{X_{N-1}} + \overline{C}) + \\ & (\overline{X_{N-1}} \cdot \overline{C}) \text{ is true} \\ N-1, & \text{otherwise} \end{cases} \quad (1)$$

$$N_{v-pq} = X_N + X_{N-1} \quad (2)$$

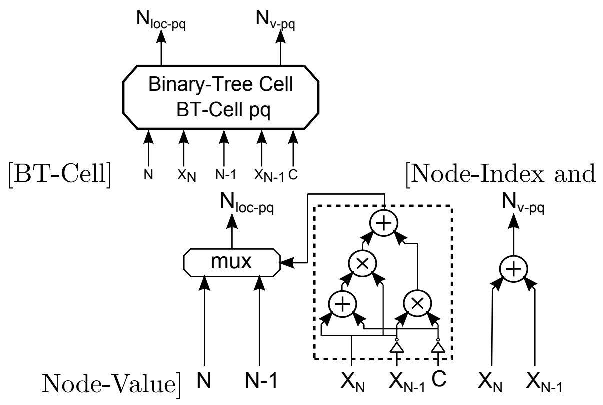


Fig.2: Binary-Tree Cell and internal logical architecture

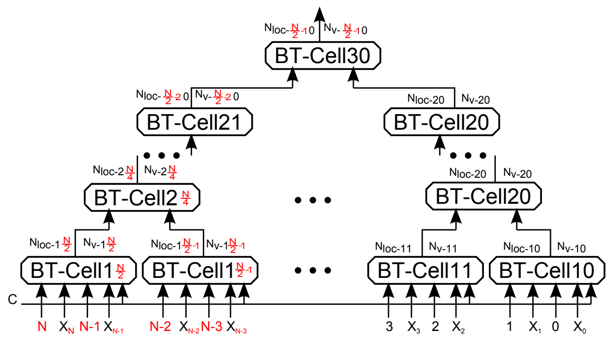


Fig.3: Binary-Tree structure

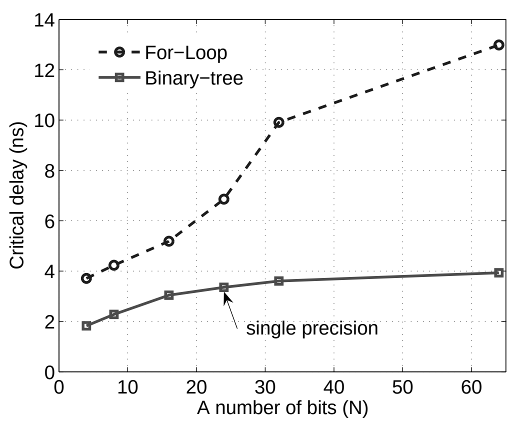


Fig.4: Performance comparison between For-loop method and Binary-Tree method

Fig. 4 depicts the performance of LOD performed by the For-Loop method compared to the Binary-Tree method, where the length of the input binary string (N) varies from 5 to 64. The graph shows that when the N increases , the critical delay of the For-Loop method dramatically increases. On the other hand, the critical delay of the Binary-Tree increases only slightly. Therefore, LOD based on the Binary-Tree architecture improves the performance of the floating-point unit.

4.2 Function Right/Left Shift

The function is normally performed by a sequential shift-register where the shifting length and the shifting direction are configurable. Similarly to the critical delay analysis of LOD, the critical delay of the sequential shift-register is proportional to the maximum shifting length of a mantissa faction. In order to alleviate this critical delay, a multiplexer-based shift-register is proposed. Assuming that n is the maximum shifting length of the registers A and B, m is a location of the LOD in the register A and B. sel is an intermediate shifting length where the shifting length is greater or equal to $n-1$. The number of utilized multiplexers is equal to n. Thus, the Right Shift (RMux) function and the Left Shift (LMux) function can be described by Eqs. (3) and (4).

$$b(m) = RMux_x = \begin{cases} a(0) & sel = 0 \\ a(1) & sel = 1 \\ \vdots & \vdots \\ a(n-1) & sel = n-m-1 \\ 0 & sel > n-m-1 \end{cases} \tag{3}$$

$$b(m) = LMux_x = \begin{cases} a(m) & sel = 0 \\ a(m-1) & sel = 1 \\ \vdots & \vdots \\ a(n-m-1) & sel = m-1 \\ 0 & sel > m-1 \end{cases} \tag{4}$$

Fig. 5 shows the performance of the multiplexer-based shift function when the shifting length varies from 5 to 64 bits. The critical delay of the sequential shift-register is comprised between 4.5 ns to 6.2 ns whereas the Multiplexer-based one maintains the critical delay between 4 ns and 4.5 ns. Thus, the Multiplexer-based shift-register achieves a higher performance than the the sequential shifting method.

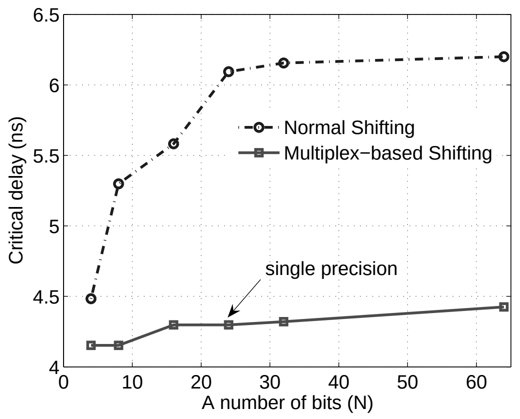


Fig.5: *Performance comparison between the Multiplexer-based and Sequential method for the Shift function*

4.3 Partial Linear Integer Multiplier based on a pipelined Architecture

Since the main critical delay of the floating-point multiplier, the floating-point PoS, and the floating-point SoP comes from an integer multiplier, improving this delay also becomes the objective of this section. The partial linear integer multiplier technique is applied to the multiplier's nominator. The pipeline architecture is utilized to improve the performance of the multiplier based on the amount of pipeline states. n and m are denominated as the number of bits of the denominator and the number of partition. The partial linear integer multiplier based on the pipeline architecture is illustrated in Fig. 6 with $m = 3$. The minimum number of pipeline states is equal to $m+1$. Carry-Ripple-Adders (CRA) are employed to add the results from each partial multiplier generated by the previous state.

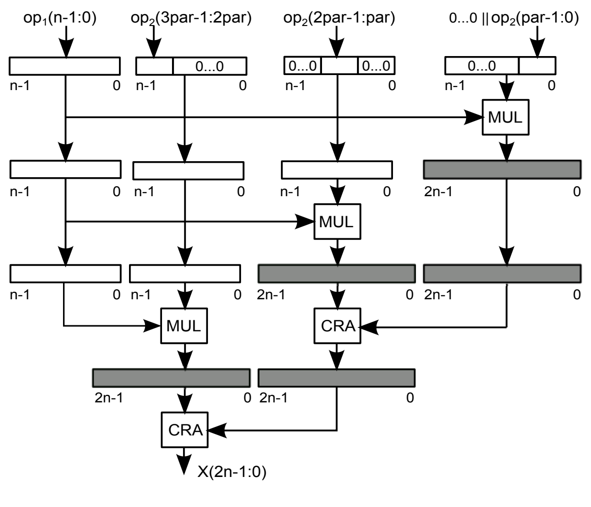


Fig.6: *A 3-partition linear integer multiplier*

Table 3: *Synthesis results of the partial linear integer multiplier on Xilinx Virtex 5 XC5VLX100t-3FF1136.*

Resources	m=1	m=2	m=3	m=4
Slice Reg.	83	146	316	466
Slice LUTs	886	894	893	905
Critical Delay(ns)	9.781	5.529	4.503	4.412
Max. Frequency(MHz)	102.23	180.86	222.08	226.63

As reported in Table 3, the pipelined partial linear integer multiplier is synthesized in order to illustrate the relationship of the consumed resources and the critical delay.

5. IMPLEMENTATION AND INVESTIGATION OF FLOATING-POINT OPERATORS

In this section, the implementation and accuracy analysis of the floating-point operational algorithms proposed in Alg. 5-8 are illustrated. Based on a pipelined architecture, the synthesis results on Xilinx Virtex 5 FPGA and 130-nm silicon technology are reported in Table 11-14. The details of partitioning states used for consideration on each operators is explained below:

5.1 Synthesis Result corresponding to state Numbers

5.1.1 FP-Adder

From Alg. 5 having 5 steps, performance and efficiency in architecture point of view by merging or separating the steps are analyzed, where the FP-Adder is investigated in 4-, 5-, and 6-state respectively. In 4-state, the step 1 and 2 of Alg. 5 are merged together. In 6-state, LOD and normalization on step 5 are separated.

5.1.2 FP-Multiplier

From Alg. 6, there are 3 cases for evaluation which are 4-,5-, and 6-state. For all cases, the step 1-3 have been grouped because their total complexity is lower than the integer multiplier's one. Thus, the 3-state FP-Multiplier becomes an initial model. To improve its performance, the integer multiplier is split as 2- and 3-state.

5.1.3 FP-PoS

From Alg. 7 which provides 7 steps, since the 1^{st} to 3^{rd} steps are grouped together, the 5-state FP-SoP becomes an initial model for consideration. As an integer multiplier generates the longest critical path, the multiplier is partitioned by two and three. Thus, there are 3 cases for evaluation 5-, 6-, and 7-state.

5.1.4 FP-SoP

Like FP-PoS, the 1^{st} to 3^{rd} steps are grouped together Alg.8 and an integer multiplier is partitioned by two and three. There are also 3 cases for evaluation 5-, 6-, and 7-state.

5.2 Comparison and Statistical Analysis in Accuracy

In this section the proposed 5-state FP-Adder method (Alg. 5) is implemented by using VHDL and then synthesized based on Xilinx Virtex IIP xc2vp30-7ff896 FPGA technology. The synthesis result is compared with 5-state FP-Adder corresponding to the methods in [19] and [20]. As illustrated in Table 4, the proposed FP-Adder provides better area and time efficiency than the two existing FP-Adder methods.

Table 4: *Area & Time efficiency of a 5-state FP-Adder.*

Module	Clock speed (MHz)	Area (Slices)
Xilinx IP [?]	120	510
FP-Adder [?]	127	394
Proposed FP-Adder	140	326

In addition, area and time efficiency of the proposed LOD method are compared with the LOD methods in [20]. The comparison result is shown in Table 5, where the proposed LOD based on binary-tree method presents better efficiency than the current LOD method proposed in [20].

Table 5: *Area & Time efficiency of LOD.*

Module	Critical Delay (ns)	Area (Slices)
LOD [?]	8.32	14
Proposed LOD	5.726	147

To compare with the 3-state FP-Adder method proposed by [21], our FP-Adder method can also provide the 3-state FP-Adder by merging step 1 to 3 of Alg. 5. However, since the FP-Adder method in [21] is designed based on Leading-Zero-Anticipator (LZA) and implemented in 0.5 μm CMOS technology which is relative older, it is not convenient for comparison with our proposed FP-Adder method.

Table 6: *Statistical error comparison of hardware float-point simulation and Matlab/Simulink, where all input operands are varied from $-10^{38.532}$ to $10^{38.532}$.*

Operator	Max. ε	Min. ε	$\lvert\bar{\varepsilon}\rvert$	$\sigma(\varepsilon)$
FP-Adder	1.0571E-6	3.7213E-12	1.6678E-7	1.6528E-7
FP-Mult.	1.4687E-6	3.8625E-15	1.5056E-7	1.9464E-7
FP-PoS	1.3892E-3	7.9421E-13	1.7340E-4	2.0198E-4
FP-SoP	3.5168E-4	6.8965E-10	4.6439E-5	4.4634E-5

For the computation precision analysis, the floating-point arithmetic units is implemented in VHDL conforming to the proposed Alg. 5-8. The results from the hardware VHDL simulation are compared the ideal results from Matlab/Simulink. The computation errors are considered and reported in statistical terms. The maximum error (Max. ε), the minimum error (Min. ε), the absolute error ($|\bar{\varepsilon}|$), and the standard deviation ($\sigma(\varepsilon)$) are listed in Table 6. For the testing environment, the value of three input operands 1 varied from $-10^{38.532}$ to $10^{38.532}$.

6. DESIGN AND ARCHITECTURE OF FLOATING-POINT ARITHMETIC ACCELERATOR

The floating-point operators, FP-Adder, FP-Multiplier, FP-PoS, and FP-SoP, which have been designed and analyzed in the previous sections are combined into a floating-point accelerator. The architecture of the accelerator is illustrated in Fig. 7. The integration of the accelerator into a multi-processor system is depicted in Fig. 8.

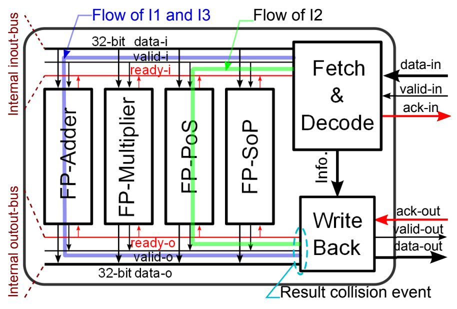


Fig. 7: *The architecture of the floating-point accelerator*

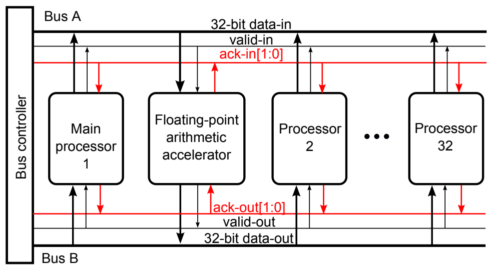


Fig. 8: *A multi-processor system integrating the floating-point accelerator*

6.1 Design and Architecture

In Fig. 7, a 32-bit pipelined instruction architecture is employed for the design of the floating-point accelerator in order to fetch and execute an intermediate instruction at every clock cycle. The design is comprised of three states : Fetch/Decode, Execute and Writeback. The signals $data - in/out$, $valid - in/out$ and $ack - in/out$ are defined as external inputs and outputs connected to BusA and BusB. The 2-bit $ack - in/out$ signal is used to notify the received status to a source module, where b'00 shows the status that the destination is in ready state ; b'01, b'10, and b'11, inform that the 1st, 2nd, and 3rd words respectively are accepted by the destination. The internal $ready-i/o$ signal indicates that the destination is in the ready state. Similarly to the handshaking protocol, a 1-word signal, $data-in/out$, coming simultaneously with a valid signal, $valid-in/out$ signal as the timing in Fig. 11 and 12. Fig. 8 shows the diagram where the proposed floating-point accelerator is applied to multi-processor system. The processors can send and receive data to and from the accelerator via a bus-system, Bus A and Bus B. A bus controller is used to handle any requests to the two buses at runtime.

The proposed floating-point accelerator is targeted to operate at the maximum frequency on Xilinx Virtex 5 FPGA (200 Mhz) and at 1 GHz using the 130-nm Silicon technology. Consequently, the corresponding number of states employed in the design of FP-Adder, FP-Multiplier, FP-PoS, and FP-SoP are 5-state, 5-state, 7-state, and 7-state respectively, in accordance to the synthesis results given in Table 11-14

6.2 Micro-Instruction and Timing Diagram

The micro-instruction pattern and the timing of the accelerator are designed in such away that it will be easily adapted to any general purpose processors. From the proposed floating-point operators, three types of instruction format, short and long (#F1 and #F2) and write-back (#R1), are introduced (Fig 9). The short and long instruction formats are respectively 3 and 4 words long. The 1st word consists of a 16-bit command (cmd), 8-bit instruction ID(I_{id}) and 5-bit processor ID (P_{id}). Up to 32 processors can be thus supported. The 2nd to 4th words are the three operands of the floating-point operation. There are 2 words for the reply format #R1 where the 1st word is composed of 8-bit instruction ID (I_{id}) and 5-bit processor ID (P_{id}). The last word is an intermediate result performed by the floating-point units. Table 7 represents the four micro-instruction table to be used by any general purpose processors.

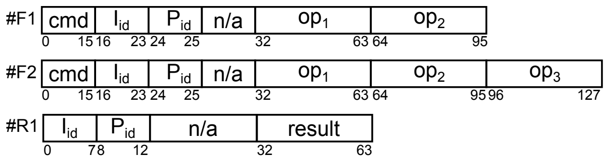


Fig.9: *Instruction format #F1, #F2 and Reply format #R1 of the accelerator*

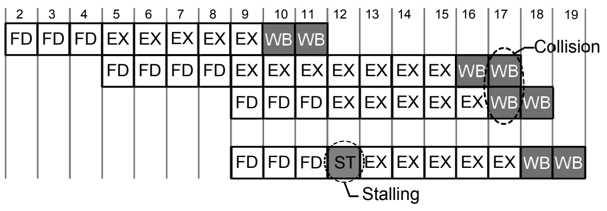

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
FD FD FD EX EX EX EX EX WB WB
FD FD FD FD EX EX EX EX EX EX EX WB WB
FD FD FD EX EX EX EX EX WB WB
FD FD FD ST EX EX EX EX EX WB WB
Collision
Stalling

Fig.10: *Result collision when either FPPoS32 or FPSoP32 instruction are first required and followed by either FPADD32 or FPMUL32, FD, EX, and WB are Fetch&Decode cycle, Execution cycle, and Write-back cycle.*

Fig. 10 shows the result of a collision event at the internal output-bus when a long instruction is ahead of a short instruction. FPADD32, FPPoS32, and FPMUL32 are called in sequence starting from the 2nd clock cycle. The result collision event happens in the 17th clock cycle, where two word results performed by FPPoS32 and FPMUL32 have been written back simultaneously. In order to alleviate this problem, a simple pattern instruction format detection is introduced into the Fetch&Decode module. If the previous instruction is a long instruction and the current instruction is a short instruction, a stalling state (ST) is used. The resulting timing diagram using the ST state is illustrated in Fig. 10, where the 1st and 2nd WB generated from the FPMUL32 are presented at the 18th and 19th clock cycle.

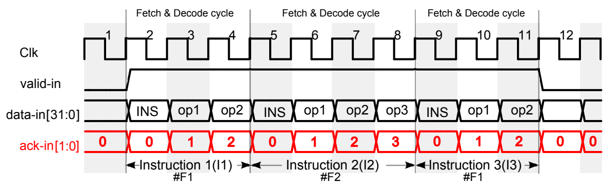


Fig.11: *The timing diagram shows the handling of three instructions by the Fetch&Decode module*

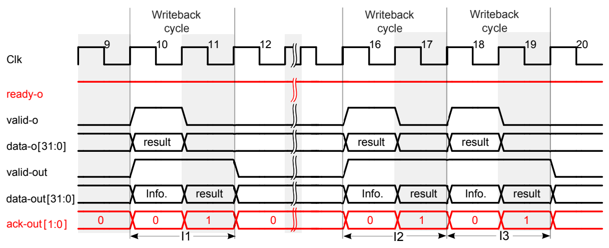


Fig.12: *The timing diagram of the personal information and the computation result of I1, I2, and I3 on their Writeback cycle.*

The timing diagram in Fig. 11 shows the execution of 3 instructions, I1, I2, and I3 which are FPADD32, FPPoS32, and FPMUL32 respectively. Each word presented by the data-in signal will be validated by the valid-in signal. Whenever destination has already received the computation result, the 2-bit $ack-in$

Table 7: *The micro-instruction of the proposed floating-point accelerator available for any general purpose processors.*

Cmd	Mnemonic	Operand	Operation	Description	#Clock
x'0001	FPADD32	I_d, P_{id}, op_1, op_2, R	$R \leftarrow op_1 + op_2$	32-bit floating-point addition	10
x'0002	FPMUL32	I_d, P_{id}, op_1, op_2, R	$R \leftarrow op_1 \times op_2$	32-bit floating-point multiplication	10
x'0003	FPPoS32	I_d, P_{id}, op_1, op_2, op_3, R	$R \leftarrow (op_1 + op_2) \times op_3$	32-bit floating-point Product-of-Sum	13
x'0004	FPSoP32	I_d, P_{id}, op_1, op_2, op_3, R	$R \leftarrow (op_1 \times op_2) + op_3$	32-bit floating-point Sum-of-Product	13

signal will by incremented. It is then reset when the result is completely received.

The timing diagram in Fig. 12 shows the Write-back cycle of the personal information (Info.) and the computation result generated by I1, I2, and I3. With always active the $ready-o$ signal, the computation result and its validation are presented on the $data-o$ signal and on the $valid-o$ signal at 10^{th}, 16^{th}, and 18^{th} clock cycle. Considering at 10^{th} clock cycle, as soon as, the $valid-o$ signal is active, the personal information of I1 is written to the $data-out$ signal, connected to the 32-bit $data-out$ signal on Bus B. It is followed by the computation result on the next clock cycle, where the $valid-o$ signal is also active for two clock cycle. In common with the writeback cycle of I1, the personal Info. and the computation result of I2 and I3 are presented at the 16^{th} and 18^{th} clock cycle.

6.3 Performance Analysis

The performance of the proposed floating-point accelerator can be evaluated by Fetch Instruction Rate (instr./s) and Throughput in number of floating-point operations per sec (FLOPS). f is denoted the maximum operating frequency of the design. The Fetch Instruction Rate (FR) means a number of floating-point instructions that can be fetched and decoded in a certain period. Obviously, the FR depends on accelerator's architecture, where if system data-width equals to the defined data-word, the FR will equal to f. However, in our design the accelerator has 32 bits data-width, where 1 data-word is 32 bits. There are two types of input instruction formats #F1 and #F2 which provide 3 and 4 data-words for one floating-point operation. Thus, the maximum and minimum FR are determined by $f/3$ and $f/4$.

Table 8 shows the Fetch Instruction Rate and the Throughput of the proposed design based on FPGA and 130-nm silicon technology. The table shows that by selecting 5-state FP-Adder and FP-Multiplier as well as 7-state FP-PoS and FP-SoP, the input rate and output rate of the accelerator are similar.

Table 8: *Performance definition and evaluation on Xilinx FPGA and 130-nm silicon technology*

Measurement		FPGA	Silicon
Max. Fetch Rate (Minstr./s)	$f/3$	66.67	333.33
Min. Fetch Rate (Minstr./s)	$f/4$	50	250
Max. Throughput (MFLOPS)	Max. FR	66.67	333.33
Min. Throughput (MFLOPS)	Min. FR	50	250

Table 9 and 10 summarizes the consumed resources and areas of the floating-point accelerator when synthesized on a Xilinx xc5vlx110t-3ff-1136 FPGA and on a 130-nm Silicon technology targeting at 200 MHz and at 1 GHz respectively.

Table 9: *Synthesis results on a Xilinx Virtex 5 device xc5vlx110t-3ff-1136.*

	Utilization	% of Total
Slice registers	1973	2%
Slice LUTs	4946	7%
Slice LUT-FF	1374	24%
BUFG/BUFGCTRLs	32	3%
Critical Delay	5 ns	
Maximum Frequency	200 MHz	

Table 10: *Synthesis result on 130-nm silicon technology.*

	Utilization
Area(um)	189513
Power(mW)	60.607
Critical Delay	1 ns
Maximum Frequency	1 GHz)

7. SUMMARY AND CONCLUSION

This paper proposes the design and analysis of floating-point operators and accelerator architecture in compliance to the IEEE standard 32-bit single-precision format. The operators are considered in their algorithmic form for hardware simplification. The standard and non-standard floating-point operators, i.e., FP-Adder, FP-Multiplier, FP-PoS, and FP-SoP are analyzed in order to increase their performance and efficiency. The PoS and SoP algorithms are also introduced by the fusion of the floating-point addition/subtration and multiplication algorithms. The Leading-One-Detection based on Binary-Tree architecture and the multiplexer-based Right/Left Shift method are proposed to alleviate the restriction derived from the maximum critical delay corresponding to the longest critical path. The partial architecture of integer multiplier is introduced and analyzed in order to improve the performance. The floating-point algorithms are implemented, synthesized, and simulated based on the hardware models in VHDL. Their computation accuracy are statistically compared with the ideal results from Matlab/Simulink. The results show that the proposed operators provide a high performance and high accuracy.

Moreover, the floating-point accelerator is designed by grouping the introduced floating-point operators. The maximum operating frequency at 200 MHz on Xilinx FPGA Virtex 5 xc5vlx110t-3ff-1136 and 1 GHz on 130-nm Silicon technology become the design constraint. In order to simplify for any general purpose processors, the micro-instruction set is introduced, where its maximum and minimum clock delay at 10 and 13 clock cycles for the short instruction format #F1 and the long instruction format #F2 are reported. From evaluation results, the accelerator provides the maximum and minimum instruction rate at 66.67 and at 50 Minstr./s on FPGA and at 333.33 and at 250 Minstr./s on Silicon. Its maximum and minimum throughput are at 66.67 and at 50 MFLOPS on FPGA and at 333.33 and at 250 MFLOPS on silicon-based technology.

References

[1] J. Gray, F. Grenzow, and M. Siedband, “Applying a PC accelerator board for medical imaging,” *IEEE Engineering in Medicine and Biology Magazine*, vol. 9, pp. 61-63, 1990.

[2] B. Bomar and B. Winkleman, “A method for accelerating the design of optimal linear-phase FIR digital filters,” *IEEE Transactions on Signal Processing*, vol. 39, no. 6, pp. 1419-1421, June 1991.

[3] C. Huntsman and D. Cawthron, “The MC68881 Floating-point Coprocessor,” *IEEE Micro*, vol. 3, pp. 44-54, 1983.

[4] C. Rowen, M. Johnson, and P. Ries, “The MIPS R3010 floating-point coprocessor,” *IEEE Micro*, vol. 8, pp. 53-63, June 1985.

[5] P. Maurer, “Design verification of the WE 32106 math accelerator unit,” *IEEE Design & Test of Computers*, vol. 5, pp. 11-21, 1988.

[6] T. Nakayama, H. Harigai, S. Kojima, H. Kaneko, H. Igarashi, T. Toba, Y. Yamagami, and Y. Yano, “A 6.7-MFLOPS floating-point coprocessor with vector/matrix instructions,” *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1324-1330, October 1989.

[7] S. Kawasaki, M. Watabe, and S. Morinaga, “A floating-point VLSI chip for the TRON architecture: an architecture for reliable numerical programming,” *IEEE Micro*, vol. 9, pp. 26-44, 1989.

[8] M. Darley, B. Kronlage, D. Bural, B. Churchill, D. Pulling, P. Wang, R. Iwamoto, and L. Yang, “The TMS390C602A floating-point coprocessor for Sparc systems,” *IEEE Micro*, pp. 36-47, 1990.

[9] F. Mayer-Lindenberg and V. Beller, “An FPGAbased floating-point processor array supporting a high-precision dot product,” in *IEEE International Conference on Field Programmable Technology*, pp. 317-320, 2006.

[10] A. Nielsen, D. Matula, C. Lyu, and G. Even, “An IEEE compliant floating-point adder that conforms with the pipeline packet-forwarding paradigm,” *IEEE Transactions on Computers*, vol. 49, no. 1, pp. 33-47, January 2000.

[11] C. Chen, L.-A. Chen, and J.-R. Cheng, “Architectureal design of a fast floating-point multiplication-add fused unit using signed-digit addition,” *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 4, pp. 113-120, July 2002.

[12] J. Bruguera and T. Lang, “Leading-one prediction with concurrent position correction,” *IEEE Transactions on Computers*, vol. 48, no. 10, pp.1083-1097, October 1999.

[13] H. Suzuki, H. Morinake, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, “Leading-Zero Anticipatory Logic for High Speed Floating-Point Addition,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 8, pp. 1157-1164, 1996.

[14] E. Hokenek and R. Montoye, “Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit,” *IBM Journal of Research and Development*, pp. 71-77, 1990.

[15] M. S. Schmookler and K. J. Nowka, “Leading Zero Anticipation and Detection A Comparison of Methods,” in *Proceedings. 15th IEEE Symposium on Computer Arithmetic*, pp. 7-12, 2001.

[16] P. Surapong, M. Glesner, and H. Klingbeil, “Implementation of realtime pipeline-folding 64-tap filter on FPGA,” in *PhD-Forum: PhD Research in Microelectronics and Electronics (PRIME)*, pp. 1-4, 2010.

[17] K. Donghyun and K. Lee-Sup, “A Floating-Point Unit for 4D Vector Inner Product with Reduced Latency,” *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 890-901, July 2009.

[18] P. Surapong and M. Glesner, “On-chip efficient Round-Robin scheduler for high-speed interconnection,” in *22nd IEEE International Symposium on Rapid System Prototyping*, pp. 199-202, 2011.

[19] W. Ligon, S. McMillan, G. Monn, F. Stivers, and K. Underwood, “A revaluation of the practicality of floating-point operations on FPGAs,” in *IEEE Symp. FPGAs for Custom Computing Machines*, pp. 206215, April 1998.

[20] A. Malik, D. Chen, Y. Choi, M. H. Lee, and S.B. Ko, “Design tradeoff analysis of floating-point adders in FPGAs,” *Canadian Journal of Electrical and Computer Engineering*, vol. 33, pp.

Table 11: *Hardware synthesis results of FP-Adder and FP-Multiplier on Vertex 5 XC5VLX100t-3FF1136.*

Operator	FP-Adder			FP-Multiplier		
Resources	4-state	5-state	6-state	4-state	5-state	6-state
Slice Reg.	200	241	290	332	395	599
Slice LUTs	442	483	488	1,132	1,159	1,155
LUT-FF pairs	187	200	255	209	253	282
Critical Delay(ns)	4.375	3.605	3.168	5.620	4.503	4.412
Max. Freq.(MHz)	228.60	277.40	315.63	177.95	222.09	226.63

Table 12: *Hardware synthesis results of FP-PoS and FP-SoP on Vertex 5 XC5VLX100t-3FF1136.*

Operator	FP-Product-of-Sum			FP-Sum-of-Product		
Resources	5-state	6-state	7-state	5-state	6-state	7-state
Slice Reg.	317	435	500	430	492	783
Slice LUTs	1642	1438	1463	1,574	1669	1634
LUT-FF pairs	260	306	337	297	325	422
Critical Delay(ns)	6.827	5.620	4.95	5.962	5.662	4.937
Max. Freq.(MHz)	146.48	177.95	202.23	167.73	177.95	202.54

Table 13: *Hardware synthesis results of FP-Adder and FP-Multiplier on 130-nm silicon technology.*

Operator	FP-Adder			FP-Multiplier		
Resources	4-state	5-state	6-state	4-state	5-state	6-state
Area(um).	16,747	18,226	19,396	39,487	40,413	47,900
Power(mW)	8.5772	10.318	18.791	14.5492	16.377	21.783
Critical Delay(ns)	0.82	0.8	0.48	0.95	0.9	0.83
Max. Freq.(GHz)	1.22	1.25	2.083	1.05	1.11	1.20

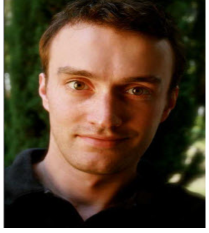
Table 14: *Hardware synthesis results of FP-PoS and FP-SoP on 130-nm silicon technology.*

Operator	FP-Product-of-Sum			FP-Sum-of-Product		
Resources	5-state	6-state	7-state	5-state	6-state	7-state
Area(um).	44,807	52,104	54,897	49,114	53,470	63,799
Power(mW)	16.99	24.923	25.716	14.692	15.510	26.193
Critical Delay(ns)	1.12	0.9	0.85	1.41	1.2	0.95
Max. Freq.(GHz)	0.89	1.11	1.18	0.71	0.83	1.05

169-175, 2008.

[21] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim, “Reduced latency IEEE floating-point standard adder architectures,” in *14th IEEE Symposium on Computer Arithmetic Proceedings*, pp. 35-42, 1999.

Pongyupinpanich Surapong was born in Prachinburi, Thailand. He received his Bachelor and Master of Engineering degree in Electrical Engineering from King Mongkut’s Institute of Technology Ladkrabang (KMITL), Thailand in 1998 and 2002. Currently, he is working toward the PhD degree in Microelectronic Systems Research Group, Technische Universität Darmstadt, Darmstadt, Germany. His research interests include computer-aided VLSI design, design optimization algorithm, circuit simulation, digital signal processing, system-on-chip, all in the context of field-programmable gate-array devices and VLSI technology.



François Philipp was born in Forbach, France. In 2009, he received a double degree from the Ecole Nationale Supérieure de l’Electronique et de ses Applications (ENSEA), Cergy, France and from the Technische Universität Darmstadt, Germany in the field of computer engineering. Since 2009, he is a Ph.D. candidate in the Microelectronic Systems Research Group at Technische Universität Darmstadt. He is working in the LOEWE-Zentrum AdRIA (Adaptronik-Research, Innovation, Application) and in the EU FP7 project MoDe (Maintenance on Demand). His research interests include acoustic signal processing, wireless sensor networks and reconfigurable hardware.

Faizal Arya Samman was born in Makassar, Indonesia. In 1999, he received his Bachelor of Engineering degree from Universitas Gadjah Mada, in Yogyakarta, Indonesia. In 2002, he received his Master of Engineering degree from Institute Teknologi Bandung, in Indonesia with Scholarship Award from Indonesian Ministry of National Education. In 2002, he was appointed to be a research and teaching staff at Universitas Hasanuddin, in Makassar, Indonesia. He received his PhD degree in 2010 at Technische Universität Darmstadt, in Germany with scholarship award from Deutscher Akademischer Austausch-Dienst (DAAD, German Academic Exchange Service). He is now working as a postdoctoral fellow in LOEWE-Zentrum AdRIA (Adaptronik-Research, Innovation, Application) within the research cooperation framework between Technische Universität Darmstadt and Fraunhofer Institute LBF in Darmstadt. His research interests include network-on-chip (NoC) microarchitecture, NoC-based multiprocessor system-on-chip, design and implementation of analog and digital electronic circuits for control system applications on FPGA/ASIC as well as energy harvesting systems and wireless sensor networks.

Manfred Glesner received the diploma degree and the Ph.D. degree from Universität des Saarlandes, Saarbrücken, Germany, in 1969 and 1975, respectively. His doctoral research was based on the application of nonlinear optimization techniques in computer-aided design of electronic circuits. He received three Doctor Honoris Causa degrees from Tallinn Technical University, Tallinn, Estonia, in 1996, Poly-technical University of Bucharest, Bucharest, Romania, in 1997, and Mongolian Technical University, Ulan Bator, Mongolia, in 2006. Between 1969 and 1971, he has researched work in radar signal development for the Fraunhofer Institute in Werthoven/Bonn, Germany. From 1975 to 1981, he was a Lecturer in the areas of electronics and CAD with Saarland University. In 1981, he was appointed as an Associate Professor in electrical engineering with the Darmstadt University of Technology, Darmstadt, Germany, where, in 1989, he was appointed as a Full Professor for microelectronic system design. His current research interests include advanced design and CAD for micro- and nanoelectronic circuits, reconfigurable computing systems and architectures, organic circuit design, RFID design, mixed-signal circuit design, and process variations robust circuit design. With the EU-based TEMPUS initiative, he built up several microelectronic design centers in Eastern Europe. Between 1990 and 2006, he acted as a speaker of two DFG-funded graduate schools. Dr. Glesner is a member of several technical societies and he is active in organizing international conferences. Since 2003, he has been the vice-president of the German Information Technology Society (ITS) in VDE and also a member of the DFG decision board for electronic semiconductors, components, and integrated systems. He was a recipient of the honor/decoration of "Palmes Academiques" in the order of Chevalier by the French Minister of National Education (Paris) for distinguished work in the field of education in 2007/2008.