LOW COST FLOATING-POINT UNIT DESIGN FOR AUDIO APPLICATIONS

Sung-Won Lee and In-Cheol Park

Division of Electrical Engineering, Department of EECS, KAIST
373-1 Gusong-dong Yusong-gu, Taejon, 305-701, KOREA

ABSTRACT

This paper presents a low-cost, single-cycle floating-point unit developed for digital audio processing applications. In the unit, the serial steps of floating-point operations are paralleled to reduce critical path delay, and hardware resources are shared with the integer datapath to minimize area overhead. Its area overhead is as small as 38% of the fixed-point datapath, and the critical path delay is 6ns in 0.25μm CMOS technology, which makes it well suited to modern audio applications such as MP3 and AAC. The floating-point unit is verified through an efficient test vector generation method developed to reduce verification time significantly.

1. INTRODUCTION

Growing demands for high quality digital audio systems have introduced several techniques. Among them, MPEG Audio Layer 3(MP3) enjoys general popularity, since music data can be compressed into 1/10 size while maintaining CD audio quality. In MP3, complex signal processing algorithms such as adaptive segmentation and sub-band filtering are employed, which require considerable processing power to achieve real time decoding of MP3 stream. So, recent implementations of MP3 decoders include a high performance DSP cores as their main processing unit. This kind of DSPs can be categorized into two types. The first is fixed-point processors [1]. Most of the audio systems use this type because the cost is low in terms of power and area. However, programmers and algorithm designers must determine the dynamic range and precision requirements either analytically or through simulation, and then add scaling operations if necessary. Therefore, the software development is generally harder than its counterpart. The second is floating-point processors [2]. The large dynamic range and precision of floating-point arithmetic make it easy to program, dispensing with scaling operations. For applications where audio quality and easy of development are more important than unit cost, floating-point processors have an advantage.

To combine the merits of both processors, that is, low cost and ease of software development, we present in this paper a floating-point unit that consumes small area. It supports three fundamental floating-point arithmetic operations including addition, subtraction, and multiplication [4] based on a 24-bit fixed-point datapath, which has the same bit-width as the popular commercial DSPs used for audio applications [8]. The floating-point format employed in our floating-point unit is different from that of IEEE 754 standard to facilitate the resource sharing with the existing fixed-point datapath. A single-cycle operation and short delay time are chosen as design constraints. The former enables to retain the current pipeline architecture developed for fixed-point operations, and the latter impedes the system performance degradation.

The organization of this paper is as follows. In Section II, floating-point arithmetic is presented, and the implementations of the proposed floating-point adder/subtractor and multiplier are described in Section III. The efficient verification scheme is discussed in Section IV, and results are given in Section V. Finally conclusions are made in Section VI.

2. FLOATING-POINT ARITHMETIC

The floating-point format consists of three fields: an exponent field, a single-bit sign field, and a fraction field. The sign field and fraction field can be considered as one unit and referred to as the mantissa field. Unlike IEEE 754 [5] in which a sign field is found at first and followed by an exponent field and a fraction field, an exponent field, a sign field and a fraction field are located in sequence.

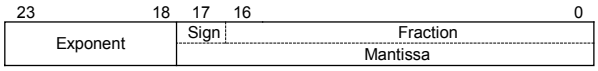

Fig. 1 Floating-point format

Fig. 1 illustrates the floating-point format that will be used throughout this paper, which conforms to the precision requirements in [3]. The value in a floating-point number x is defined as follows

$$x = m \cdot 2^{e}$$
$$where,\ m = \begin{cases} 01.f_2 = 1 + f & if\ s = 0 \\ 10.f_2 = -2 + f & if\ s = 1 \end{cases} \quad (1)$$

In (1), s is the value of the sign bit, f_2 is the binary value of the 17-bit fraction field, and e is the signed decimal value of the 6-bit exponent field. The mantissa m represents a normalized 2's-complement number, that is, the most significant nonsign bit is implied. Thus it provides additional precision of one-bit. With this definition, the mantissa is in the interval $m \in [1, 2)$ if $s=0$, and $m \in [-2, -1)$ if $s=1$. The reserved value in (2) is used to represent zero.

$$e = -2^{b_e - 1} = (100000)_2 \qquad s = 0$$
$$f = 0 = (0\_0000\_0000\_0000\_0000)_2, \quad (2)$$

where b_e is 6, the number of bits in the exponent fields.

We examined the required dynamic range for the MP3 encoder to check whether the proposed floating-point format is suitable or not. At least 180dB of dynamic range is required to make MP3 audio stream without audio quality degradation. The dynamic range provided by the proposed format satisfies this re-

quirement. Table 1 summarizes the range and precision of the floating-point format.

TABLE 1 Range and precision of the floating-point format

Case	Value	D.R.
Most positive	$(2-2^{-17})\times 2^{31} = 4.294950912\times 10^{9}$	190dB
Least positive	$1\times 2^{-31} = 4.656612873\times 10^{-10}$	
Least negative	$(-1-2^{-17})\times 2^{-31} = -4.656648400\times 10^{-10}$	
Most negative	$-2\times 2^{31} = -4.294967296\times 10^{9}$	

Numbers not in range are either underflowed or overflowed. If an underflow occurs, we set the number to 0, and if an overflow occurs, it is replaced by the most positive or negative value according to its sign. Special symbols of the IEEE 754 such as NaNs, $+\infty$, or $-\infty$ are not considered for the sake of compact implementations. For the same reason, truncation is chosen as the rounding strategy.

In this paper, floating-point addition, subtraction and multiplication are considered, because they are intensively used in digital signal processing applications. To define the floating-point operations, let us consider two floating-point numbers, $a = m_a \times 2^{e_a}$ and $b = m_b \times 2^{e_b}$.

The sum (or difference) of a and b, s is defined as

$$\begin{aligned} s &= a \pm b \\ &= (m_a \pm m_b \times 2^{-(e_a - e_b)}) \times 2^{e_a}, if\ e_a \geq e_b \\ &= (m_a \times 2^{-(e_b - e_a)} \pm m_b) \times 2^{e_b}, if\ e_a < e_b \end{aligned} \quad (3)$$

The product of a and b, p is defined as

$$\begin{aligned} p &= a \times b = m_a \times m_b \times 2^{(e_a + e_b)} \\ thus,\ m_c &= m_a \times m_b \\ e_c &= e_a + e_b \end{aligned} \quad (4)$$

In implementing the operations shown above, the 2's-complement mantissa representation makes a chance to share the fixed-point datapath without any format conversion. This is particularly beneficial in multiplication as the multiplier takes considerable area. Therefore we can implement the floating-point unit with small area overhead.

3. IMPLEMENTATION

Typical fixed-point DSP processors are composed of 5 stage pipelines: instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (EX) and write back (WB). Memory access arises in IF, OF and WB stage, and complex address generation occurs in ID stage, and long arithmetic operations such as multiplication happens in EX stage. Therefore overall clock speed usually does not exceed 200MHz.

In order to augment a floating-point unit to the fixed-point processor, two conditions must be satisfied. First, the floating-point unit must complete its operation within one-cycle. Multi-cycle instructions usually induce more complex control circuitry, hence invoking big changes in the control unit, and causing difficult data forwarding and pipeline control. Second, it must be fast. If the processor is extremely slowed down by inserting the floating-point unit, such modification cannot be accepted. In Fig. 2 the overall architecture is presented. Two floating-point modules are inserted into execution stages to support three fundamental floating-point operations.

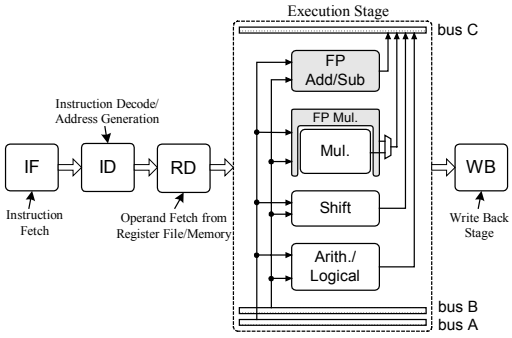

Fig. 2 Overall Architecture

3.1 Floating-Point Addition and Subtraction (FPAS)

FPAS consists of three steps: mantissa alignment, mantissa computation (add or subtract) and exception handling. Each step also contains a few small operations. Most of VLSI Implementations [6,7,9] of FPAS adopt two or more stages pipeline for fast execution, but we cannot use it to maintain single-cycle execution. Though each step for the proposed floating-point format is simpler than that of IEEE 754, the serial execution delay of all three steps is still long, that is to say, about 10ns. In order to minimize the delay of the single-cycle FPAS, we have optimized the first two steps. As the last step takes less time and is relatively simple, we exclude it from optimization.

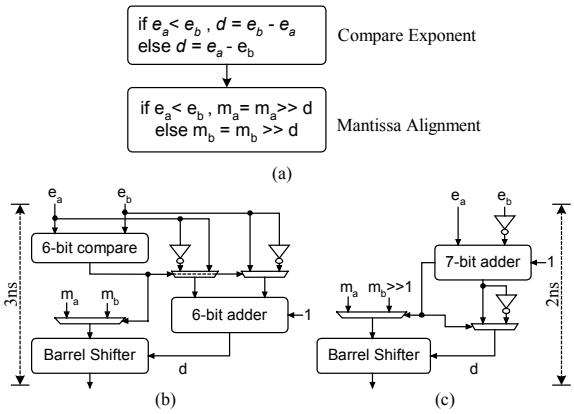

Fig. 3 Mantissa alignment step for FPAS: (a) operation description, (b) serial implementation, (c) proposed implementation

The mantissa alignment step is composed of comparing exponent and aligning mantissa as shown in Fig. 3(a). After comparing two exponents, we know which one is larger. Then the mantissa that has smaller exponent is shifted to right arithmetically by the difference, d between the larger and the smaller. The serial implementation of these is illustrated in Fig. 3(b), and 3ns is measured for its delay. For the purpose of shortening the delay, we replace the 6-bit comparator and the 6-bit adder by one 7-bit adder and two inverters. Indifferent to the serial implementation, we subtract e_b from e_a in advance to determining which one is larger. If e_b is larger, i.e. the sign bit of the 7-bit adder is one, the difference is inverted. However this value is smaller than the correct positive value by one, so we shift m_b to right by one to

solve this problem. The reason that the 7-bit adder replaces the 6-bit adder is the sign bit of the 7-bit adder is utilized in multiplexers to select a proper one and the overflow check circuitry usually has longer delay than the sum of a 1-bit adder. Fig. 3(c) illustrates this structure that achieves 1ns delay enhancement.

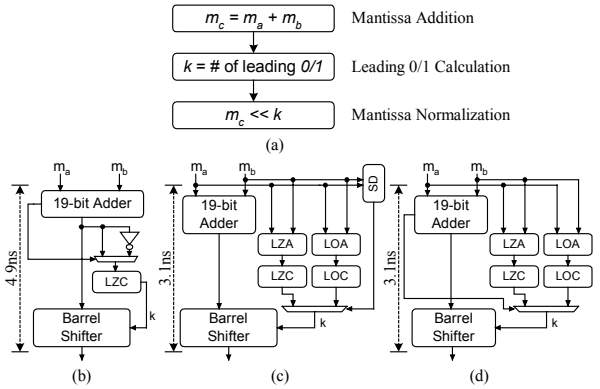

Fig. 4 Mantissa computation step for FPAS: (a) operation description, (b) serial implementation with LZC[6], (c) LZA with SD[7], (d) proposed implementation

The next-step, mantissa computation consists of three small operations as shown in Fig. 4(a). After mantissa addition, we need the leading-sign-bit (LSB) count to normalize m_c. For the serial implementation of Fig. 4(b), we always set the result positive. Then the number of the LSBs is calculated using a leading-zero-counter (LZC) [6]. Using the value as the input of the barrel shifter, the normalization is accomplished. As a result, 4.9ns is measured for delay. To make the 19-bit addition and the LSB count be parallel, the leading-zero-anticipator (LZA) [6] was introduced. The sign-detector (SD) [7] for the 19-bit addition was presented as well. When the length of addition is very long, the SD works pretty well since the shift amount of the barrel shifter cannot be prepared until addition is completed. However the delay of 19-bit addition is close to that of LZA plus LZC, so the SD can induce area overhead without any delay reduction. The mantissa computation step based on the SD and the LZA is shown in Fig. 4(c), and the proposed implementation that does not include the SD is depicted in Fig. 4(d). The leading-one-anticipator (LOA) and leading-one-counter (LOC) is also used for the negative result. The LOA can be implemented by adding an inverter to each input of the LZA. Though both cases reduce the delay to 3.1 ns, we choose the latter that requires less area.

In order to support the subtract operation, we have to decide where is the right place to toggle the sign of an operand. In definition (3), m_b is subtracted from m_a after two mantissas are aligned. This can be implemented using one multiplexer and one inverter next to m_b, and another carry-in control signal for the 19-bit adder in the mantissa computation step. This method has a problem that the multiplexer and the inverter are inserted in the critical path, leading to additional delay for the FPAS. We solve it by moving the multiplexer and the inverter to the next of m_b in the mantissa alignment step. This approach does not change the critical path delay of FPAS because the select signal of the multiplexer is arrived after m_b has been modified. The equivalence between two approaches is shown in (5).

$$\overline{m_b \cdot 2^{-d_e}} = -2 \cdot \overline{m_{b,18}} + \sum_{i=18-d_e}^{17} \overline{m_{b,18}} \cdot 2^{i-17} + \sum_{i=d_e}^{17} \overline{m_{b,i}} \cdot 2^{i-17-d_e} + \cancel{\sum_{i=0}^{d_e-1} \overline{m_{b,i}} \cdot 2^{i-17-d_e}}$$
$$\overline{m_b} \cdot 2^{-d_e} = -2 \cdot \overline{m_{b,18}} + \sum_{i=18-d_e}^{17} \overline{m_{b,18}} \cdot 2^{i-17} + \sum_{i=d_e}^{17} \overline{m_{b,i}} \cdot 2^{i-17-d_e} + \cancel{\sum_{i=0}^{d_e-1} \overline{m_{b,i}} \cdot 2^{i-17-d_e}}$$
$$\therefore \overline{m_b \cdot 2^{-d_e}} = \overline{m_b} \cdot 2^{-d_e}, \; where \; d_e = e_a - e_b > 0 \quad (5)$$

The last step checks special cases (underflow, overflow, zero) and converts the result to the appropriate value if required. Due to the proposed computation steps, 6.0ns is obtained for the overall delay for FPAS.

3.2 Floating-Point Multiplication (FPMUL)

Like FPAS, FPMUL consists of three steps. The first step is mantissa multiplication and exponent addition, and the second is mantissa normalization. The last step deals with the special cases.

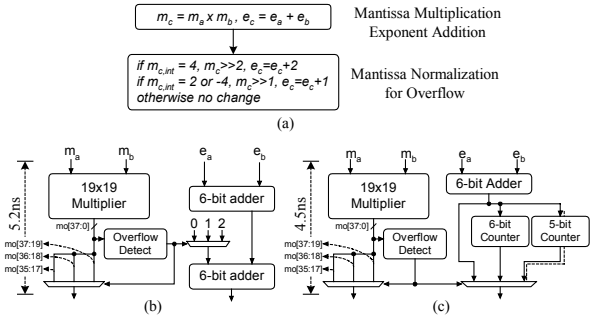

Fig. 5 First two steps for FPMUL: (a) detailed description, (b) serial implementation, (c) proposed implementation

Fig. 5(a) shows a detailed description of the first two steps. Mantissa multiplication shares the existing fixed-point multiplier, so no change allowed during this steps. However some parallelism exists in mantissa normalization step, i.e. incrementing the exponent by 1 or 2 can be done before noticing the actual value of the increment. In Fig. 5(b) the serial implementation is illustrated and its delay is 5.2ns. The proposed implementation that calculate exponent early is shown in Fig. 5(c). The 6-bit counter is for incrementing by 1, and the 5-bit counter is for adding by 2. With a little area overhead, we achieve 4.5ns delay. The remaining parts detect the exceptions such as overflow, underflow, and zero, and perform the corresponding actions.

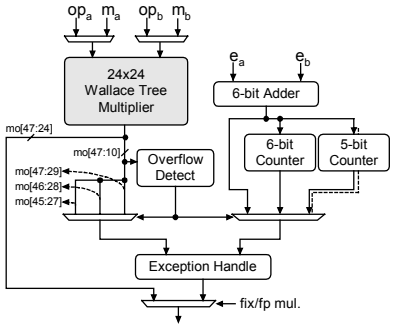

Fig. 6 FPMUL implementation using the existing 24×24 fixed-point multiplier

Fig 6 illustrates the final FPMUL implementation. Two multiplexers are inserted at the input of the multiplier to select a proper one from a fixed-point operand and a floating-point mantissa. Another multiplexers are placed at the end to choose one of two outputs. The critical path delay for FPMUL is 5.4ns where the fixed-point multiplier produces the correct answer in 4.2ns.

4. VERIFICATION

A reference model, which has been verified completely, plays and important role design verification. As the floating-point arithmetic considered in this paper is different from others, we build the reference model from scratch in C language. In order to examine whether the model works correctly or not, many test vectors are required. The random vector generation scheme [10] can be used to test the operation of the fabricated chip, but it is not acceptable to verify a software implementation that takes extremely long simulation time. Design errors are hard to found using ordinary test vectors that do not generate any exceptions. For the above two reasons, we discover that the random vector generation using LFSRs is an inefficient method.

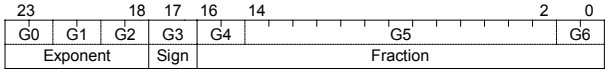

Fig. 7 Bit grouping for the test vector generation

To make test vectors include exceptions as many as possible, we propose a new scheme based on the observation that many errors are related to zero, overflow, and underflow occurrence. Overflow is probable to appear when two similar numbers of large absolute value are calculated, and for two numbers that have small exponent and similar mantissa, underflow is expected. If two numbers are the same except the sign part, or one operand is zero in multiplication, zero is generated. As shown in Fig. 7, three groups are allocated for the exponent. This results in similar exponents for two operands. One group is used for the sign field to generate zero cases. Another three groups are used for the fraction. This also produces similar fraction values. Using this scheme, 16384 pairs are generated to verify each floating-point operation, and we can debug the model quickly.

TABLE 2 Synthesis results

Fixed-point datapath	Shifter	804 gates
	ALU	1238 gates
	Multiplier	7959 gates
	Total gates	10001 gates
	Critical path delay	4.2ns (Multiplier)
Floating-point datapath	FPMUL	1241 gates
	FPAS	2567 gates
	Total gates	3808 gates (38% overhead)
	Critical path delay	6.0ns (FPAS, 43% overhead)

5. RESULTS

The fixed-point processor including the proposed floating-point unit is synthesized with UMC 0.25μm cell library at typical operating condition. Synthesis results for each datapath are summarized in Table 2. Adding the floating-point results 12% area overhead with respect to the entire processor and 38% with respect to the fixed-point datapath. In Table 3, the performance of the proposed floating-point units is compared to other floating-point processors designed for audio applications in the literature.

TABLE 3 Performance comparison

	Delay (ns)	Area (gates)	Technology	Pipeline stages
Tsou [11]	20	N/A	0.6μm	1
Yamada [9]	6.7	N/A	0.3μm	2
IEEE754	7.5	26093	0.35μm	3
This	6.0	3808	0.25μm	1

6. CONCLUSIONS

A low-cost, single-cycle floating-point unit has been presented in this paper, which is developed to support audio applications requiring more dynamic range and precision. Three fundamental floating-point operations are supported for easy software development by eliminating the careful scaling and rounding that is indispensable if only fixed-point operations are available. The area cost is low, as the proposed floating-point arithmetic can be implemented by sharing hardware resources such as the multiplier with the fixed-point unit. Single-cycle floating-point operations lead to easy integration with the existing fixed-point datapath. With 38% area overhead, we achieved critical path delay of 6ns, which is enough to support the state-of-the-art audio applications.

7. REFERENCES

[1] L. Bergher, X. Figari, F. Frederiksen, M. Froidevaux, J. Gentit, and O. Queinnec, “MPEG Audio Decoder for Consumer Applications,” Proc. of CICC, pp. 413-416, 1995.

[2] S. Hong, B. Park, Y. Song, H. Seo et al., “A Full Accuracy MPEG1 Audio Layer 3 (MP3) Decoder with Internal Data Converters,” Proc. of CICC, pp. 563-566, 2000.

[3] Greg Maturi, “Single Chip MPEG Audio Decoder,” IEEE TCE, vol. 38, no. 3, pp. 348-356, Aug. 1992.

[4] K. Kontro, K. Kallojärvi, and Y. Nuevo, “Use of Short Floating-Point Formats in Audio Applications,” IEEE TCE, vol. 38, no. 3, pp. 200-207, Aug. 1992.

[5] IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985, New York, NJ, USA, Aug. 1985.

[6] H. Suzuki, H. Morinaka, H. Makino et al., “Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition,” IEEE JSSC, vol. 31, no. 8, pp. 1157-1164, Aug. 1996.

[7] K. Lee and Kevin J. Nowka, “1GHz Leading Zero Anticipator Using Independent Sign-Bit Determination Logic,” Digest of Technical Papers SOVC, pp. 194-195, 2000.

[8] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, DSP Processor Fundamentals: Architectures and Features, IEEE Press, Piscataway, NJ, 1997.

[9] H. Yamads, T. Hotta, T. Nishiyama F. Murabayashi et al., “A 13.3ns Double-precision Floating-point ALU and Multiplier,” Proc. of ICCD, pp. 466-470, 1995.

[10] N. Sakashita, H. Sawai, E. Teraoka, T. Fujiyama et al., “Built-In Self-Test in A 24bit Floating Point Digital Processor,” Proc. of ITC, pp. 880-885, 1990.

[11] K. Tsou, O. Chen, and C. Hu, “A Cost-Effective Fixed/Floating-Point Digital Signal Processor,” Proc. of VTSA, pp. 213-216, 1997.