A 24-BIT FLOATING-POINT AUDIO DSP CONTROLLER SUPPORTING FAST EXPONENTIATION

Sung-Won Lee, Hyeong-Ju Kang, and In-Cheol Park

Department of Electrical Engineering and Computer Science, KAIST
373-1 Guseong-Dong Yuseong-Gu, Daejeon 305-701, Republic of Korea

ABSTRACT

This paper describes a 24-bit floating-point DSP controller developed for audio applications. For easy system design, a floating-point DSP and interface blocks including USB, I^2S, and UART are integrated onto a single chip. Based on the floating-point arithmetic optimized for audio processing algorithms, a fast, single-cycle floating-point unit is designed to achieve high performance. In addition, a special unit to calculate exponential function is proposed. The exponentiation unit uses three instructions to compute x^y without losing audio quality.

1. INTRODUCTION

Growing demands on high quality digital audio have introduced several new 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. The MP3 algorithm consists of several complex signal processing steps, thus most MP3 decoders require considerable processing power to operate in real-time, leading to a high performance digital signal processor (DSP) core [1]. Recently, an MP3 decoder based on a floating-point DSP was introduced [2] for the sake of full audio accuracy and easy software development. However, its 32-bit floating-point format is not optimized for audio applications even if we consider the encoder that needs more dynamic range and precision than the decoder.

This paper presents a 24-bit floating-point DSP designed for audio applications including the MP3 decoder and encoder. Although the design of a 24-bit DSP has been the subject of several papers that focused on audio applications, our approach is different, because it provides an exponentiation unit that can save the considerable memory space and the execution cycles, and the floating-point arithmetic that facilitates resource sharing with a fixed-point datapath is not used elsewhere.

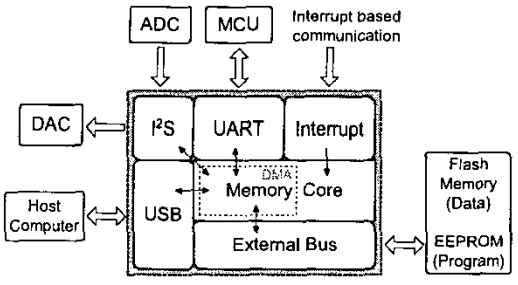


Figure 1. Input/Output interface of the audio DSP controller.

This work was supported by the Korea Science and Engineering Foundation through the MICROS center and IC Design and Education Center.

2. OVERVIEW

The I/O interface of the proposed DSP is shown in Figure 1. An analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) are connected through I^2S interface. The ADC generates raw audio bit stream and the DAC transforms decoded bit stream into analog waveform. A micro controller unit (MCU) can be connected to the DSP via UART for the auxiliary functions such as graphic user interface control. Four external interrupt pins are provided for the interrupt-based data communication. USB is used to download or upload audio files between the DSP and a host computer, and the 8-bit external bus attaches flash memory and EEPROM to the DSP. In flash memory, audio data is stored, and EEPROM holds audio processing algorithms that will be transferred to the on-chip program memory by a small bootstrap program. Most of memory transactions are controlled by a DMA controller, which means the DSP core is not concerned with I/O events.

As shown in Figure 2, the proposed DSP is composed of five stage pipelines: instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (EX), and write back (WB). In IF stage, program memory access occurs, and data memory access occurs in OF and WB stage. In ID stage, instruction decoding is followed by a memory operand address generation based on the diverse addressing mode. The register file is composed of 16-entry 24-bit registers with four read ports and two write ports. In EX stage, there are five functional units (floating-point adder/subtractor, exponentiation unit, floating-point multiplier sharing fixed-point multiplier, arithmetic/logical unit, and shifter with format conversion circuit). Two units among FP Add/Sub, FP Mul, and Arithmetic/Logical can operate in parallel using four source buses and two destination buses. All the instructions that use these five units are made to complete within one-cycle,

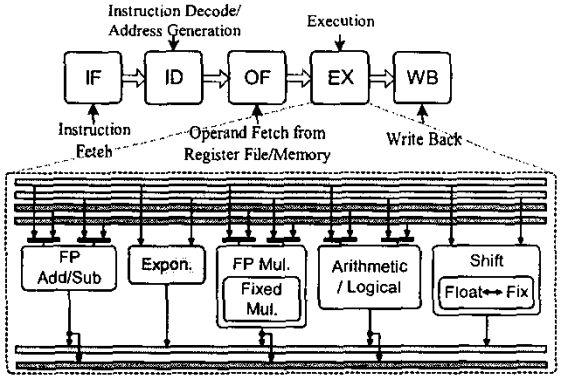


Figure 2. Overall Architecture.

because multi-cycle instructions usually induce complex control circuits and invoke additional delay penalty. Another reason is that the floating-point arithmetic employed in the proposed DSP is made to be relatively simple compared with the IEEE 754 standard [3], thus the involved floating-point operation can be implemented in a single-cycle without speed penalty.

3. 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 [3] 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 as shown in Figure 3.

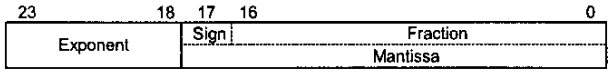


Figure 3. Floating-point format.

The value of a floating-point number x is defined as follows

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

In (1), s is the value of the sign bit, f_2 is the binary value of the 17-bit fraction field. The mantissa m represents a normalized 2's-complement number, that is, the most significant non-sign bit is implied. Thus it provides additional precision of 1-bit. The reserved value in (2) is used to represent zero.

$$e = -32,\ s = 0,\ f = 0 \tag{2}$$

The floating-point format offers 180dB of dynamic range that satisfies the requirement to encode high quality MP3 audio files. Table 1 summarizes the range and precision of the floating-point format. Numbers not in range are either underflowed or overflowed. If an underflow occurs, we set the number to zero, 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, +∞, and -∞ are not considered for the sake of compact implementation. For the same reason, truncation is chosen as the rounding scheme.

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

$$c = a \pm b = (m_a \pm m_b \times 2^{-(e_a - e_b)}) \times 2^{e_a} \quad if\ e_a \geq e_b$$
$$= (m_a \times 2^{-(e_b - e_a)} \pm m_b) \times 2^{e_b} \quad if\ e_a < e_b \tag{3}$$

The product of a and b is defined as

$$c = a \times b = m_a \times m_b \times 2^{(e_a + e_b)}$$
$$Therefore,\ m_c = m_a \times m_b,\ e_c = e_a + e_b \tag{4}$$

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	1×2^{-31}	$= 4.656612873\times 10^{-10}$	
Least negative	$(-1-2^{-17})\times 2^{-31}$	$= -4.656648400\times 10^{-10}$	
Most negative	-2×2^{31}	$= -4.294967296\times 10^{9}$	

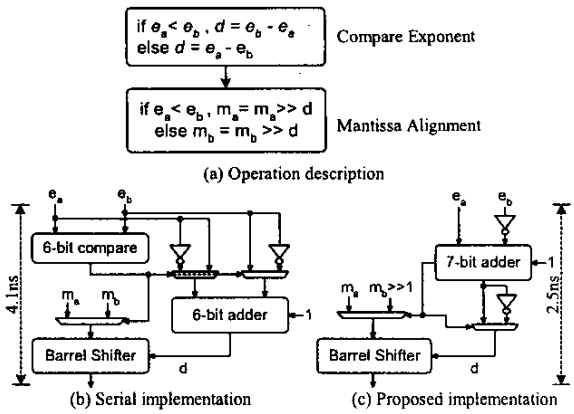


Figure 4. Mantissa alignment step of FPAS.

3.1 Floating-Point Adder and Subtractor (FPAS)

FPAS consists of three steps: mantissa alignment, mantissa computation (add or subtract) and exception handling. To achieve fast single-cycle FPAS computation, we have optimized the first two steps, since the last step takes less time and is relatively simple. For the rest of paper, we use 0.35μm cell library for delay estimation.

The mantissa alignment step is decomposed into comparing exponent and aligning mantissa as shown in Figure 4(a). By comparing two exponents, we can determine the larger one. The mantissa with the smaller exponent is shifted right arithmetically using d, the difference of two mantissas. The serial implementation in Figure 4(b) has 4.1ns delay. To improve the delay, we replace the 6-bit comparator / adder by one 7-bit adder and two inverters. Unlike the serial implementation, we subtract e_b from e_a before determining the larger exponent. If e_b is larger, the difference is inverted. Since this value is still smaller than the correct positive value by one, we shift m_b right by 1 bit. The 6-bit adder is replaced by the 7-bit adder for simple overflow detection. Figure 4(c) illustrates this structure that achieves 1.6ns delay enhancement.

The next-step, mantissa computation, is decomposed into three small operations as shown in Figure 5(a). After mantissa addition, we need the leading-sign-bit (LSB) count to normalize m_c. The serial implementation in Figure 5(b) always sets the result positive. Then, a leading-zero-counter (LZC) [4] counts the number of leading zeros. Using this value as the input, the barrel shifter normalizes m_c in 6.4ns. To make mantissa addition and LSB counting parallel, a leading-zero-anticipator (LZA) was introduced [4]. A sign-detector (SD) to determine the barrel shifter shift amount earlier was also presented [5]. If operands of addition are very long, the SD works pretty well. However, the delay of 19-bit addition is close to that of LZA plus LZC. Therefore the SD induces area overhead with little delay reduction. The mantissa computation step with the SD and the LZA is shown in Figure 5(c), and the proposed implementation not having SD is shown in Figure 5(d). The leading-one-anticipator (LOA) and leading-one-counter (LOC) are employed for the negative results. The LOA is implemented by adding an inverter to each input of the LZA. The proposed implementation reduces the delay to 4.1 ns. Consequently, the overall delay of FPAS is 7.9ns.

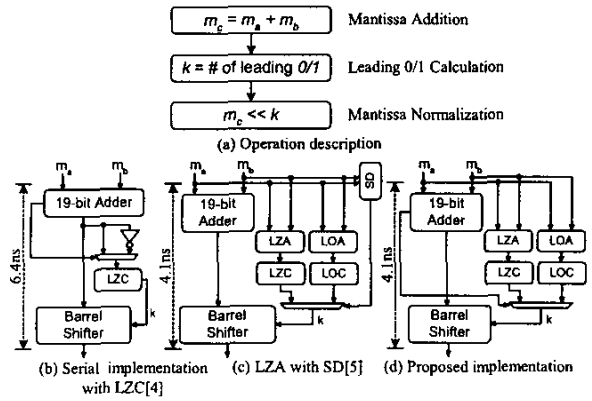


Figure 5. Mantissa computation step of FPAS.

3.2 Floating-Point Multiplier (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 special cases. Figure 6(a) shows a detailed description of the first two steps. Mantissa multiplication shares the existing fixed-point multiplier, so no change is allowed for this step. 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 Figure 6(b) the serial implementation is illustrated and the delay is 6.9ns. The proposed implementation that calculates an exponent earlier is shown in Figure 6(c). The 6-bit and 5-bit counters are employed to increment by 1 and 2 respectively. With a little area overhead, we achieve 5.9ns delay. The critical path delay of FPMUL is 7.2ns where the delay of the fixed-point multiplier is 5.5ns.

4. EXPONENTIATION UNIT

In the digital audio algorithm, especially MP3 encoding algorithm, several key steps such as quantization and psycho-acoustic modeling require exponentiation functions. As power is not restricted to integer values, a series of multiplication cannot be employed. As a result, a complex library function such as *pow()* or a table that has pre-computed values is normally used instead. The first method using a library function occupies less memory space than the table-lookup method (function resides in program

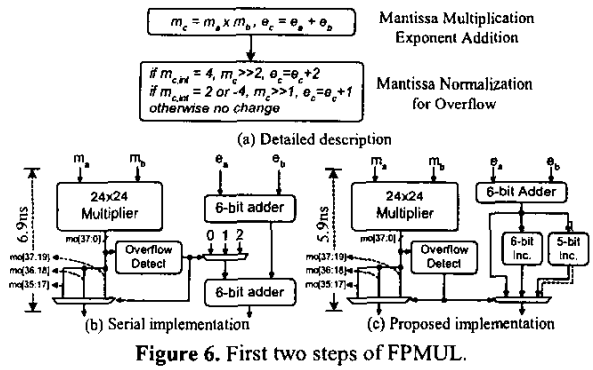


Figure 6. First two steps of FPMUL.

memory and table is in data memory), but it takes long execution cycles. This implies fast computation is hard to achieve. The second table-lookup method gains speed-up at the cost of large memory space. If the table size can be managed to be small, it is very efficient. Modern audio applications, however, need approximately 10K~20K table entries, which mandates large memory and complex addressing. To combine the advantages of both methods, we propose an approximated exponentiation. Though the similar approach has been proposed in image processing applications [6], our floating-point arithmetic simplifies the implementation and makes it easy to be integrated into the DSP core with negligible change of control circuits.

The exponentiation x^y can be calculated as $2^{y \cdot \log_2(x)}$, which can be computed by a sequence of the logarithmic, multiplication, and exponential operations. A piece-wise linear function is used to approximate $log_2(x)$ as shown in (5). Compared to the approximations of [6], ours is more complex, but more accurate. As there exists more complex and time-consuming blocks such as FPAS, we focus on the accuracy.

$$\log_2(x) = \log_2(2^e \cdot m) = \log_2(2^e \cdot (1+f)) \cong \begin{cases} e + 41/32 \cdot f & \text{if } 0.00 \le f < 0.25 \\ e + 135/128 \cdot f + 15/256 & \text{if } 0.25 \le f < 0.50 \\ e + 57/64 \cdot f + 9/64 & \text{if } 0.50 \le f < 0.75 \\ e + 98/128 \cdot f + 30/128 & \text{if } 0.75 \le f < 1.00 \end{cases} \quad (5)$$

In (5) x is assumed to be a floating-point number. To implement the circuit, we use another notation as in (6). The fixed-point representation with the decimal point at the 18th bit position is employed to represent the result because the normalized floating-point format usually degrades the precision of the result.

$$\log_2(x) \cong \begin{cases} e.0 + f + f >> 2 + f >> 5 + 0 & \text{if } 0.00 \le f < 0.25 \\ e.0 + f + f >> 4 - f >> 7 + 15/256 & \text{if } 0.25 \le f < 0.50 \\ e.0 + f - f >> 3 + f >> 6 + 9/64 & \text{if } 0.50 \le f < 0.75 \\ e.0 + f - f >> 3 + f >> 6 + 15/64 & \text{if } 0.75 \le f < 1.00 \end{cases} \quad (6)$$

As the dynamic range of y is restricted, y has the same format as the result of $log_2(x)$. The multiplication between y and $log_2(x)$ is computed in the 24x24 multiplier. Unlike normal multiplication that truncates the lower 24 bits, we truncate the lower 18 bits to preserve the location of the decimal point. 2^x is also approximated as shown in (7).

$$2^x = 2^{e+f} \ (e: \text{integer}, 0 \le f < 1) \cong \begin{cases} 2^e \cdot (1 + 97/128 \cdot f) & \text{if } 0.00 \le f < 0.25 \\ 2^e \cdot (1 + 29/32 \cdot f - 9/256) & \text{if } 0.25 \le f < 0.50 \\ 2^e \cdot (1 + 9/128 \cdot f + 7/8) & \text{if } 0.50 \le f < 0.75 \\ 2^e \cdot (1 + 17/64 \cdot f + 47/64) & \text{if } 0.75 \le f < 1.00 \end{cases} \quad (7)$$

The result conforms to (1), so it can be stored in the floating-point format. As the floating-point format assumes a leading 1, the leading 1 in the mantissa is removed in implementation. Finally, we use the notation shown in (8)

$$2^f = \begin{cases} f - f >> 2 + f >> 7 + 0 & \text{if } 0.00 \le f < 0.25 \\ f - f >> 3 + f >> 5 - 9/256 & \text{if } 0.25 \le f < 0.50 \\ f + f >> 4 + f >> 7 - 1/8 & \text{if } 0.50 \le f < 0.75 \\ f + f >> 2 + f >> 6 - 17/64 & \text{if } 0.75 \le f < 1.00 \end{cases} \quad (8)$$

The linear approximation of 2^x and $log_2(x)$ are composed of similar computations, so we make both of the approximations have a single exponentiation unit. Figure 7 shows the structure of the exponentiation unit. By employing this unit, x^y can be calculated

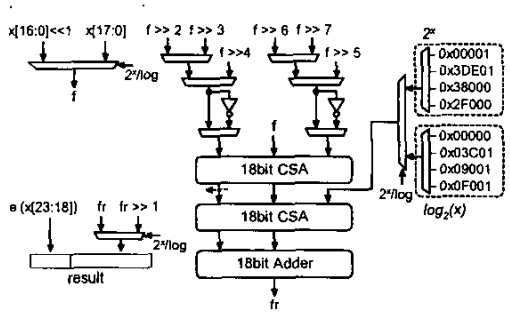


Figure 7. Structure of the proposed exponentiation unit.

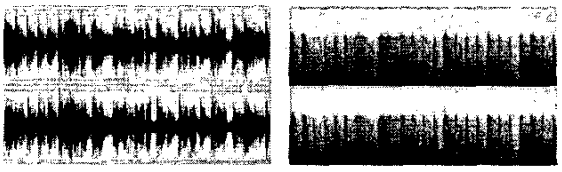
with three instructions: $log_2(x)$, modified multiply, and 2^x. It requires no extra memory space that is indispensable for the table-lookup approach.

5. RESULTS

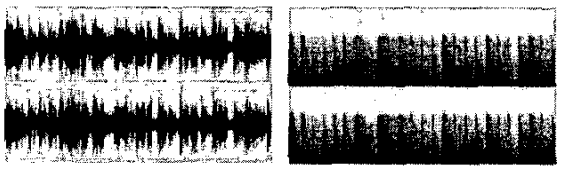
To investigate the effects of error incurred by the exponentiation unit, we have applied our approximation algorithm to the MP3 encoder. The approximation algorithm replaces the library function, *pow()*. In Figure 8, the decoded results are compared. Figure 8(a) shows the result from a reference encoder using library pow() function and Figure 8(b) shows the result from a modified encoder using the proposed exponentiation algorithm. In each figure, the left is a time domain waveform and the right is a frequency-domain spectrum. For time-domain the difference is quite negligible and for frequency-domain a few differences are found in the high frequency region.

In addition, decoded wave files are compared to source wave files in terms of a peak error and a RMS error. As shown in Figure 9 the reference encoder and the proposed encoder demonstrate quite similar peak error performance and almost same RMS error performance.

Figure 10 shows the layout of the proposed floating-point DSP controller that equips the proposed exponentiation unit. It has about 145,000 gates and the size is 2.88mm × 2.84mm in 0.35μm QLM CMOS technology. The processor operates at 80MHz clock frequency.



(a) Reference implementation using Spare's *pow()*



(b) With proposed exponentiation algorithm

Figure 8. Performance evaluation of the approximated exponentiation.

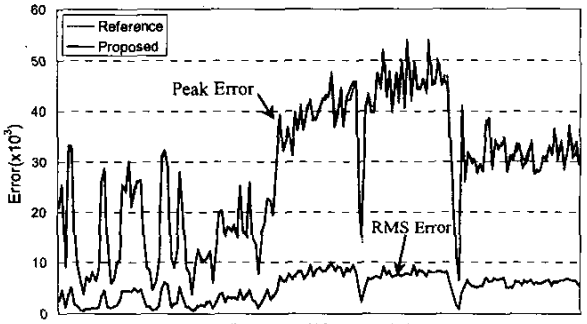


Figure 9. Error performance comparison.

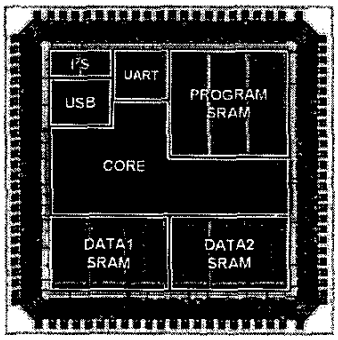


Figure 10. Layout of the proposed audio DSP controller.

6. CONCLUSIONS

This paper has described a 24-bit floating-point DSP controller specialized for audio applications. Many interface blocks were integrated to assist a rapid design of an audio system such as MP3 codec. Based on the floating-point arithmetic optimized for audio processing algorithms, a fast, single-cycle floating-point unit was designed, and an exponentiation unit that can calculate x^y with only three instructions was proposed for the fast and memory efficient implementation with negligible loss of accuracy.

7. REFRENCES

[1] L. Berghcr, X. Figari, F. Frederiksen, M. Froidevaux, J. Gentit, and O. Queinnec, "MPEG Audio Decoder for Consumer Applications," in *Proc 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," in *Proc CICC*, pp. 563-566, 2000.

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

[4] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase et al., "Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition," *IEEE JSSC*, vol. 31, pp. 1157-1164, Aug. 1996.

[5] K. Lee and K. J. Nowka, "1GHz Leading Zero Anticipator Using Independent Sign-Bit Determination Logic," in *SOVC Dig. Tech. Papers*, pp. 194-195, 2000.

[6] S. Nam, B. Kim, Y. Im, Y. Kwon et al., "FLOVA: A Four-issue VLIW Geometry Processor with SIMD Instructions and Lighting Acceleration Unit," in *Proc CICC*, pp. 551-554, 2000.