Digital Audio Effect System-on-a-Chip Based on Embedded DSP Core

Kyungjin Byun, Young-Su Kwon, Seongmo Park, and Nak-Woong Eum

I. Introduction

Over the last few decades, digital audio effects have been developed for various audio and music applications, such as composition, recording, mixing, and real-time sound processing. A number of implementation techniques for audio effects are used, such as filters, delay lines, time-segments, and time-frequency representations [1]. Although audio effect algorithms and their realization by software have been extensively studied [2]-[6], few studies have been devoted to the complete system implementation of audio effects [7], [8].

Using the hardware/software co-design approach described in [7] provides a practical alternative to software centric systems. In [7], dedicated hardware units are used to implement real-time audio effects while software units are used for parameterization of the effects, which controls and manages the hardware effect units. The software units are implemented on an embedded soft-core processor supplied by the field programmable gate array (FPGA) vendor. The advantages of the implementation of audio effects using a dedicated hardware are very high throughput and low latency. However, implementing audio effects in hardware is more difficult and time-consuming than in software. Because the hardware and software units are implemented together on an FPGA operating at 48 MHz in [7], the performance of the implemented units is relatively lower than that of a fabricated system-on-a-chip (SoC).

Implementation of a professional audio effect system by using a commercial digital signal processor (DSP) is described in [8]. The authors realized a total of nine audio effects in real-time in a software centric manner. They insisted that using DSPs is the most efficient implementation solution in signal-based applications involving digital processing because DSPs feature high performance and flexibility at acceptable costs.
However, because their implementation is not an SoC, it requires other components, such as an audio codec, in order to implement a complete audio effect system.

In this paper, we present the implementation of a digital audio effect SoC which integrates an embedded DSP (eDSP) core, audio codec IP, a number of peripheral blocks, and various audio effect algorithms. Generally, digital audio effects are realized by software in almost all cases. However, in this study, we developed the audio effect SoC using a software and hardware co-design method. The embedded DSP and some dedicated hardware blocks are developed as a hardware design, while the audio effect algorithms are realized in a software centric manner.

The design of the eDSP and its development environments are also described in this paper, which were developed by Electronics and Telecommunications Research Institute (ETRI) for various audio and speech applications, such as an audio codec [9] and a video codec [10]. One of the useful features of the eDSP compiler is its use of primitive functions. If a user modifies application programs written in C language by using the primitive functions of the eDSP compiler, efficiently optimized compilation results can be obtained. As a result, employing the primitive functions makes it possible to obtain an optimized assembly code because our compiler is fully customized to the eDSP.

In the implementation of audio effects, we employ the primitive functions of the eDSP compiler, which provide an efficient way to implement the audio effect algorithms. Most of the algorithms are realized by C language with the primitive functions that run on the embedded DSP in real-time, while the equalizer, which requires a large amount of computing power, is implemented by the dedicated hardware block with high flexibility.

Section II describes the design of the embedded DSP core, its development environments, and the primitive functions of the compiler. In section III, we present the digital audio effect algorithms and an efficient implementation method for them. Then, the design of the audio effect SoC and its implementation results are presented in section IV. Section V gives the conclusion of this paper.

II. Design of Embedded DSP Core

1. Embedded DSP Core

The eDSP is a 16-bit fixed-point DSP, which was developed by ETRI for use in an SoC suitable for audio and speech applications. The eDSP can consist of a single, dual, or quad core data processing engine (DPE) because it has a scalable architecture. Each core of the eDSP efficiently shares the memory module via an access arbiter [11]. This architecture is very efficient for developing SoCs for various applications that need a different level of computation and resources. The main features of the eDSP core shown in Fig. 1 are the following:

- Five functional units: I (instruction sequencing unit), A (data address generation unit), X (execution unit), M (memory access unit), and P (peripheral unit)
- One program bus and three 16-bit data buses (two for data read and one for data write)
- Unified memory architecture (both the program and data share the same memory space)
- Hardware-based pipeline conflict resolving technique
- Seven stage pipeline: prefetch, fetch, decode, access, read, execute, and write
- Arithmetic instructions with 32-bit operands

A DPE consisting of I, A, and X units is the central instruction processing core of the eDSP. The DPE performs most operations of the eDSP, such as instruction fetch and decoding, address generation, and arithmetic operations. In
Table 1. Some primitive functions and their operation.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description and operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>accum_t</td>
<td>Type definition of 40-bit accumulator</td>
</tr>
<tr>
<td>extract_low</td>
<td>Extract 16-bit low word of accumulator</td>
</tr>
<tr>
<td>extract_high</td>
<td>Extract 16-bit high word of accumulator</td>
</tr>
<tr>
<td>to_high</td>
<td>Load 16-bit word to high word of accumulator</td>
</tr>
<tr>
<td>subc</td>
<td>Divide step for integer division which is repeated for 16 times for 16-bit division</td>
</tr>
<tr>
<td>fract_mpy</td>
<td>Fractional multiplication</td>
</tr>
</tbody>
</table>

Fig. 2. Modification of the division function by using the ‘subc’ primitive function.

Division (plain C code)

```c
division(var1, var2)
{
    accum_t aa = var1;
    accum_t bb = var2;
    aa = aa << 15;
    bb = bb << 15;
    for (int i = 0; i < 16; i++)
    {
        aa = subc(aa, bb);
    }
    return extract_low(aa);
}
```

Division with primitive function

```c
division(var1, var2)
{
    accum_t aa = var1;
    accum_t bb = var2;
    aa = aa << 15;
    bb = bb << 15;
    for (int i = 0; i < 16; i++)
    {
        aa = subc(aa, bb);
    }
    return extract_low(aa);
}
```

Fig. 1, the I unit arranges instructional words to handle various types of instructions. The A unit fetches operands from the memory or internal DPE registers. The X unit executes the operation and saves the results into the memory or the DPE registers. Each unit is a clearly separated module written in hardware description language (HDL) code.

The M unit is a parallel memory system that arbitrates the data requests between DPEs and memory modules. The M unit is composed of an access arbiter and the memory modules. The access arbiter arranges the memory requests from a DPE or set of DPEs. The arbiter detects conflicts among memory requests, arranges the requests, and services accesses in the order of highest priority. Conflict-generating memory requests are serviced sequentially while conflict-free requests are completed immediately.

2. Primitive Functions of eDSP Compiler

The compiler of the eDSP provides some useful primitive functions which can be used in the C application layer but they are not instructions of eDSP. Application programmers can use the primitive functions as a subfunction in their program written in C language for optimizing their codes. Therefore, if these primitive functions are used in implementing an algorithm by using C language, well optimized assembly codes can be obtained which are compilation results. We also used these primitives in the implementation of the audio effect algorithms, which will be discussed in section III. The compiler of the eDSP provides about 20 primitive functions, such as “extract_high,” “to_high,” “subc,” and “fract_mpy.” If a programmer modifies the application programs written in plain C code by using the primitive functions, very compact compilation results can be obtained, that is, very compact assembly codes. Table 1 shows some primitive functions and their operation, which are used in Figs. 2 and 7.

Therefore, we exploit these primitive functions to develop mathematical functions such as logarithm, exponential, and division functions, which are used for implementing audio effects. The logarithm function is used in distortion effects, and the division function is used in auto-wah and phaser effects, which will be discussed in section III. Figure 2 shows the division function written in C language with and without primitives.

Mathematic functions written in C with primitives show highly optimized compilation results. For instance, a logarithmic function implemented using the primitives requires a computation of less than 10% of the code compared to not using the primitives. In particular, the division function implemented by using a ‘subc’ primitive needs only the computation of 4% of the plain C code compared to not using a primitive function.

III. Digital Audio Effects

Digital audio effects are used in various audio applications, such as multi-effectors, karaoke systems, and mobile audio devices. The main goal of digital audio effects is the modification of the sound characteristic of the input audio signal. There are many audio effect algorithms such as reverb, distortion, auto-wah, and pitch shifting. These effects can be configured as shown in Fig. 3.

1. Audio Effect Algorithms

Many publications have dealt with the design of digital audio effect algorithms and their realization [5], [6]. Therefore, we briefly describe some algorithms that will be mentioned again in the following sections for the implementation issues.

A. Reverb Effect

Reverberation is a very common phenomenon in our lives. A reverb effect is the result of many reflections of sound
that occur in a room and on the surrounding walls in a concert hall. From any sound source, there are direct and indirect paths. The sound through an indirect path is reflected, delayed, and attenuated. These reflected waves can again bounce off another wall before arriving at our ears, and so on. The reverb effect is the realization of a series of delayed and attenuated sound waves [1].

In our implementation, the reverb effect is realized by combining a parallel chain of comb filters (CF) and a serial chain of all-pass filters (APF), as depicted in Fig. 4. The outputs of the comb filters are summed together through an all-pass filter to produce a reverb effect.

B. Pitch Shift

A pitch shift changes the pitch of an audio signal without affecting its speed. Various techniques for pitch shifts have been proposed such as a phase vocoder and synchronous overlap-add (SOLA). Both are block-based algorithms. A phase vocoder is based on the short time Fourier transform, and the SOLA approach is a time domain block-based algorithm. These block-based algorithms have a latency problem, which is a major issue in live systems in which an audio signal or voice is pitch-shifted in real-time [6]. To avoid the latency problem, we adopted an interpolated pitch shifting method in our implementation, which is a sample-by-sample processing algorithm.

C. Distortion Effect

Distortion is the modification of a sound using nonlinear signal processing. Some musical instruments such as electronic guitars take advantage of the distortion effect to enlarge and vary their timbre. This modifies the sound color by introducing nonlinear distortion products of the input signal [1]. In our implementation, we adopted logarithm and Bezier curves for nonlinear processing to generate a distortion curve as shown in Fig. 5. A distortion effect is obtained by performing the logarithmic calculation followed by the computation of the Bezier curve. The distortion effect needs a preprocessing such as a noise gate because it amplifies the input signal very much.

D. Equalizer

Equalization is an effect that allows the frequency response of an output signal to be controlled. An equalizer produces an equalization effect that boosts or cuts certain frequency bands to adjust the output audio sound [8]. In this paper, we implement an equalizer using a filter bank composed of infinite impulse response (IIR)-type filters. Because digital filter processing requires repetitive and intensive computation, these IIR filters are implemented by a dedicated hardware block. The multiband equalizer can be realized using these filters. The filter coefficients for each IIR filter are supplied by the eDSP and are calculated in the eDSP in advance. The audio input signal is also fed into each filter by the eDSP, and the filtered output is taken by the eDSP.
and 7, QN is 14 and MAF0_FB is a constant in Q14, which functions of the eDSP, respectively. The C code used in Figs. 6 shows many kinds of intrinsic functions, most of them are C programmers in order to optimize C codes. Although intrinsic functions of C54x DSP and using the primitive compiler is fully customized to the eDSP.

It is possible to obtain an optimized assembly code because our method does not involve intrinsic functions they might rewrite their C code which seems to be almost an assembly-level code, as shown in Fig. 6, because even if those intrinsic functions can be used like a C function, they are instruction-level functions, that is, one intrinsic function is compiled to one DSP instruction. These kinds of intrinsic functions are efficient for optimizing C codes such as ETSI standard speech codecs, but not for plain C codes. In the “C code with intrinsic” in Fig. 6, the “_smac” intrinsic function of C54x DSP is used to optimize the C code, which is the same as the “MAC” instruction of C54x DSP.

On the other hand, the primitive functions of eDSP can be used like operators in a plain C code as shown in Fig. 7 because the primitive functions of eDSP are more primitive than the intrinsic functions of C54x DSP. This means the primitive functions have higher flexibility. In other words, for eDSP, programmers can use some primitive functions in order to describe MAC operation in the C code, while for C54x DSP, programmers should use the _smac intrinsic function for the MAC operation. Therefore, when optimizing the C code it is not necessary for programmers to modify their code much to use the primitive functions of eDSP. Although there are differences between the primitive function of eDSP and the intrinsic functions of C54x DSP, the results of optimization using primitive and intrinsic functions are competitive.

In the example in Fig. 7, we convert the original C code in the upper left into the different one shown in the upper right by using primitive functions, such as extract_high, to_high, and fract_mpy. Using these three primitives makes the compiler generate the optimized compiled results shown in the lower right in Fig. 7, in which a multiply and accumulate (MAC) instruction is employed by the compiler of eDSP, while the compiled results without primitives demonstrate an inefficient code as shown in the lower left of Fig. 7.

An MAC instruction is an instruction among the eDSP instruction set, and it is efficient because it performs multiplication and addition simultaneously. Therefore, letting the compiler employ a MAC instruction, as shown in the lower right of the Fig. 7, is an efficient way to reduce the code size and computation. As shown in Fig. 7, a compiled assembly code of the C code with primitive functions is more compact and less cycle-consuming than that without primitives. Figure 8 shows the procedure in which each primitive function used in Fig. 7 makes the compiler generate the optimized compilation results in which a MAC instruction is employed by the compiler.

First, as shown in Fig. 8, the to_high primitive moves the data in memory into acc_high, the high 16-bit part of the 40-bit accumulator. At this time, the low 16-bit acc_low is filled with zeros. Second, the fract_mpy primitive performs multiplication and shifts left by 2 bits. Last, the extract_high primitive moves

<table>
<thead>
<tr>
<th>Effects</th>
<th>Resource</th>
<th>C54x</th>
<th>eDSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverb</td>
<td>Program memory</td>
<td>1,642</td>
<td>1,465</td>
</tr>
<tr>
<td></td>
<td>Data memory</td>
<td>44,384</td>
<td>44,382</td>
</tr>
<tr>
<td></td>
<td>MIPS</td>
<td>40.6</td>
<td>39.6</td>
</tr>
<tr>
<td>Pitch shift</td>
<td>Program memory</td>
<td>1,094</td>
<td>1,365</td>
</tr>
<tr>
<td></td>
<td>Data memory</td>
<td>10,171</td>
<td>10,231</td>
</tr>
<tr>
<td></td>
<td>MIPS</td>
<td>5.2</td>
<td>8.5</td>
</tr>
</tbody>
</table>

Table 2. Comparison of compilation results between C54x DSP and eDSP (MIPS: million instructions per second).

<table>
<thead>
<tr>
<th>C code</th>
<th>Without intrinsic</th>
<th>With intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6. Optimization of using intrinsic function of TI C54x.

<table>
<thead>
<tr>
<th>C code</th>
<th>Without primitive</th>
<th>With primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7. Effectiveness of using primitive functions of eDSP.
Fig. 8. Procedure generating the optimized results in Fig. 7 by using the primitive functions.

Table 3. Improvements for the implementation of reverb by using the primitive functions.

<table>
<thead>
<tr>
<th></th>
<th>Without primitive</th>
<th>With primitive</th>
<th>Assembly coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory</td>
<td>1,465</td>
<td>1,099</td>
<td>851</td>
</tr>
<tr>
<td>Data memory</td>
<td>44,382</td>
<td>44,351</td>
<td>44,401</td>
</tr>
<tr>
<td>MIPS</td>
<td>39.6</td>
<td>33.6</td>
<td>24.0</td>
</tr>
</tbody>
</table>

the high 16-bit part of the accumulator into the memory. As a result, combining these primitive functions generates more optimized compilation results employing a MAC instruction.

Table 3 shows the improvement of the implementation of the reverb effect by using the primitive functions. The computation of the reverb with primitives is reduced by 15% compared to that without primitives. The rightmost results are from using assembly coding, which is the most optimized version; however, this method requires a long development time. We can choose the implementation method according to our needs. For instance, the reverb effect, which requires a large computation and memory, is implemented by assembly coding, and other algorithms are implemented using cross-compilation with the primitive functions for development within a short time.

IV. Audio Effect SoC

1. Implementation of Audio Effect SoC

Conventional audio effects are generally implemented on a microprocessor or a DSP by software. Therefore, they need some components, such as an audio codec, a phase locked loop (PLL), an analog-to-digital converter (ADC), and interface logics in order to develop audio effect systems, such as audio mixers and electric guitar effectors. We designed and implemented an audio effect SoC which integrates almost all of the following components which are needed for developing an audio effect system:

- Embedded DSP as a processor to perform various signal processing of audio effect algorithms
- Stereo audio codec IP for converting an analog audio input signal into a digital signal
- ADCs for the acquisition of the analog input for adjusting parameters of various audio effects
- External memory interface for an audio effect which requires large memory
- Boot-loading interface for downloading or uploading the firmware of audio effects from or to the external program memory space
- PLL for proper clock frequency

The audio effect SoC integrates a dual eDSP core, stereo audio codec, multiband equalizer, and many peripherals, such as a PLL, ADCs, host interface, and external memory interface as shown in Fig. 9. Because the audio effect SoC is based on the eDSP core, most of the peripheral blocks and hard-wired blocks are connected to the eDSP core through its external user (EU) bus.

We can change the parameters and configuration of the audio effects by using the host interface, because the host interface has an 8-bit bus and a memory of 256 words. It is also used to download or upload the firmware in the eDSP. The eDSP firmware is also downloaded from an external EEPROM using the boot-loader interface for the standalone mode.

Fig. 9. Block diagram of audio effect SoC.
The SoC includes an 8-channel ADC and ADC interface, which has an 8-bit resolution. The ADC is for controlling the parameters of the effect algorithms, such as reverb depth, pitch value of the pitch shifting, and noise threshold. Its maximum conversion rate is 2 MHz, which is enough to control the parameters of the audio effects. Although the ADC has one physical channel, it can be considered an 8-channel converter because it is multiplexed by an 8-channel multiplexer.

The audio codec IP is an 18-bit 48 kHz stereo sigma-delta audio codec for digital audio applications. Users can initialize the audio codec by using the serial peripheral interface (SPI). The codec interface makes it possible to use a user-specific external audio codec. When using the internal audio codec, audio data bypasses the codec interface. The external memory interface is for the external SDRAM. The SoC includes an internal memory of 128 kB, which is not enough for effects such as a delay-type algorithm. External memory can be extended to 2 MB. The PLL is an analog programmable frequency synthesizer for an on-chip application. Its output range and operating frequency is from 5 MHz to 320 MHz, and the loop characteristics of the PLL are fully programmable.

In the SoC, most effect algorithms are implemented by software as described in section III, except for the equalizer. The multiband equalizer consists of several filter blocks, which require a huge amount of computation. For instance, the computation of the equalizer is about 6 MIPS per band, resulting in 60 MIPS for a 10-band equalizer. Therefore, we realize the equalizer by employing dedicated hardware blocks that consist of universal filters and can be reconfigured by changing the filter coefficients and some parameters. Moreover, we use these filter blocks for implementing other effect algorithms, such as a distortion effect, which uses the IIR filter.

The design of the audio effect SoC is verified using the FPGA development board as shown in Fig. 12, which includes an Altera Stratix EP1S80B956C6 FPGA, an audio codec, an ADC, and a microcontroller for debugging our SoC design. In this board, we also verified the implemented audio effect algorithms in real-time before fabricating the SoC.

The audio effect SoC can be used in the many applications, such as electric guitar effectors and audio mixers. In case of implementing an electric guitar effector, just the audio effect SoC and EEPROM are needed because the SoC includes all components to perform audio effects. An example of an audio effect system is shown in Fig. 10. This example shows how the SoC can be used in an application system such as an electric guitar effector.

In Fig. 10, the audio effect firmware which is implemented by C and compiled by eDSP compiler is stored in an EEPROM. The electric guitar is used as an audio input device, and reverb effect is used as an audio effect in this example. Electric guitarists usually use many kinds of effectors when they perform. The electric guitar effector operates as follows.

After the system power is turned on, the firmware of the audio effects in the EEPROM is downloaded to the internal memory of the SoC through the boot-loading interface. The downloaded audio effect program runs on the DSP waits for the input audio signal. The analog guitar signal is converted to a digital signal by the audio codec at a rate of 44.1 kHz. The audio effect program receives a digital signal from the audio codec as an input signal. The audio effect program performs the reverb algorithm to reverberate the input signal. The reverb program must process the input samples of 44,100 in a second for the real-time processing. The output signal which is reverberated by the reverb program is converted again to an analog signal by the audio codec and is amplified for the speakers.

2. Summary of Audio Effect SoC

Table 4 provides a summary of the implemented audio effect SoC. The data memory size is 64k 16-bit words, which can be used for several effects.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual core eDSP</td>
<td>Max. 120 MIPS per core</td>
</tr>
<tr>
<td>Gate count</td>
<td>26,000 gates per one eDSP core</td>
</tr>
<tr>
<td>Memory</td>
<td>64k words for eDSP, 512 words for interface blocks</td>
</tr>
<tr>
<td>Hard-wired block</td>
<td>4 filter blocks (equalizer)</td>
</tr>
<tr>
<td>Audio codec IP</td>
<td>Stereo in/out with 18 bit resolution</td>
</tr>
<tr>
<td>Peripherals</td>
<td>PLL, ADC, external memory I/F, host I/F, boot I/F, ADC I/F, external codec I/F</td>
</tr>
<tr>
<td>Fabrication</td>
<td>0.18 µm CMOS process, 144 TQFP package, die size: 5.5 mm × 5.5 mm</td>
</tr>
<tr>
<td>Power consumption</td>
<td>Typically 168 mW (3.3V / 1.8V)</td>
</tr>
</tbody>
</table>
Table 5. Implementation results of audio effects.

<table>
<thead>
<tr>
<th>Effects</th>
<th>Prog. mem (words)</th>
<th>Data mem (words)</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverb</td>
<td>1,099</td>
<td>44,351</td>
<td>33.6</td>
</tr>
<tr>
<td>Chorus</td>
<td>386</td>
<td>2,136</td>
<td>5.8</td>
</tr>
<tr>
<td>Flanger</td>
<td>366</td>
<td>4,152</td>
<td>4.9</td>
</tr>
<tr>
<td>Phaser</td>
<td>507</td>
<td>78</td>
<td>8.1</td>
</tr>
<tr>
<td>Tremolo</td>
<td>279</td>
<td>452</td>
<td>3.0</td>
</tr>
<tr>
<td>Auto-wah</td>
<td>523</td>
<td>45</td>
<td>9.1</td>
</tr>
<tr>
<td>Pitch shift</td>
<td>573</td>
<td>3,180</td>
<td>7.0</td>
</tr>
<tr>
<td>Distortion</td>
<td>465</td>
<td>1,046</td>
<td>9.3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4,198</strong></td>
<td><strong>55,440</strong></td>
<td><strong>80.8</strong></td>
</tr>
</tbody>
</table>

The implementation results of the audio effects are summarized in Table 5. We initially implemented eight basic audio effects in our SoC, but we will add other effects or change the configuration of the effect algorithms according to the specific application because implementing the effect algorithms on the SoC is based on a programmable eDSP. As mentioned in the previous section, the SoC includes a dual core eDSP that has a computing power of 120 MIPS per core, and the implemented audio effects in Table 5 require a total computation of 81 MIPS. Therefore, there is enough room to add other complicated effect algorithms. Figure 11 shows a die photograph of the audio effect SoC, and Fig. 12 shows the SoC evaluation and FPGA development boards.

V. Conclusion

In this paper, we presented the implementation of a digital audio SoC including an embedded DSP design and the efficient realization of a number of audio effect algorithms using the primitive functions of the eDSP compiler. Besides the implemented basic effects in the SoC, we will add variations and other effects into the SoC to expand its application areas in the future. The quality of the audio effect SoC will be finally evaluated by musicians, because audio effects are usually employed by such users. The implemented SoC is suitable for application in various audio applications, such as audio mixers, guitar-effectors, karaoke systems, and mobile audio devices.

References


**Kyungjin Byun** received the BS degree in electronics engineering from Kookmin University, Seoul, Korea, in 1987, the MS and PhD degrees in electronics engineering from Information and Communications University (ICU), Daejeon, Korea, in 2000 and 2006, respectively. He was a visiting scholar at Purdue University, W. Lafayette, USA in 2007. Since 1987, he has been with Electronics and Telecommunications Research Institute (ETRI) as a principal member of research engineering staff, and has participated in various projects including the development of digital communication systems, an embedded DSP core, and speech codecs. His current research interests include speech and audio signal processing, as well as digital signal processors and their applications.

**Seongmo Park** received the BS, MS, and PhD degrees in electronics engineering from Kyungpook National University, Daegu, Korea, in 1985, 1987, and 2006, respectively. From 1987 to 1992, he was with the LG semiconductor company, Gumi, Korea, where he worked on ASIC design. In 1992, he joined Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea, where he has worked on the development of SoC design. He is currently engaged in research on SoC design, image compression algorithms, and SoC architecture design. He is now a team leader of the Multimedia Processor Design Team. He is also a professor of the University of Science and Technology. He has published over 30 technical papers in international journals and conference proceedings. His main research interests are video coding, image compression, multi-core processor design, DSP design, and low-power SoC architecture design.

**Young-Su Kwon** received the BS, MS, and PhD degrees from Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea, in 1997, 1999, and 2004, respectively. He was with Microsystems Technology Laboratory, Massachusetts Institute of Technology as a postdoctoral associate from 2004 to 2005. He has been with the System-on-Chip Research Department, Electronics and Telecommunications Research Institute, Republic of Korea, since 2005. He has authored 30 internal journal and conference papers with special interest in multi-core processor design, VLSI CAD, and algorithmic optimizations of circuits and systems.

**Nak-Woong Eum** received the BS degree in electronic engineering from Kyungpook National University, Daegu, Korea, in 1984, and the MS and PhD degrees in electrical engineering from KAIST, Korea, in 1987 and 2001, respectively. Since 1987, he has been working with ETRI, Daejeon, Korea, as a principal member of engineering staff, where he is currently the director of the System-on-Chip Research Department. His current research interests include embedded processor technology, mobile multimedia systems, and system-on-chip design methodology.