A BLOCK-FLOATING-POINT PROCESSOR FOR RAPID APPLICATION DEVELOPMENT

Hiroaki Tanaka, Yoshinori Takeuchi, Keishi Sakanushi, Masaharu Imai

Graduate School of Information Science and Technology,Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan

Shiro Kobayashi

Asahi Kasei Corporation
Okada 3050, AXT Maintower 22F
Atsugi, Kanagawa 243-0021, Japan

ABSTRACT

This paper proposes an instruction set processor which uses hierarchical block-floating-point (H-BFP) arithmetic. H-BFP has been developed to provide efficient development approach for digital signal processing system. H-BFP offers highly abstracted computation model, and leads to cost-effective implementation. However, application development based on H-BFP is still time consuming task because of lacking high level programming language. In this paper, a processor architecture together with a set of instructions to support H-BFP are proposed. We also propose a related software development environment for efficient algorithm translation which generate high performance codes without any time consuming task. Experimental results show proposed environment achieves high performance signal processing systems with H-BFP processors.

Index Terms— Digital Signal Processors, Fixed Point Arithmetic, Design Methodology

1. INTRODUCTION

Digital signal processors (DSPs) are one of the most important components to realize digital signal processing systems. DSPs are widely used to develop consumer electronic products and release them to market in a short development period. There are two major types of DSPs classified by the supported arithmetic. The first one is floating-point arithmetic and the other is fixed-point arithmetic. Each of these DSPs has different features. From the application software development point of view, software for floating-point DSPs can be efficiently developed, because high signal quality can be easily achieved due to the nature of floating-point arithmetic. However, the floating-point units are expensive and not suitable to realize cost-effective digital signal processing systems. On the other hand, fixed-point DSPs do not have expensive hardware. For the reason of low hardware cost, fixed-point DSPs are widely used in consumer electronics. However, the development of software for fixed-point DSPs is time consuming task because it is difficult to develop fixed-point implementation which achieves high signal processing quality.

As a compromise between floating-point and fixed-point, block-floating-point (BFP) implementation is also used on fixed-point DSPs. While a BFP implementation can realize high signal processing quality similar to floating-point on a low-cost fixed-point hardware, developing BFP implementation is still a hard task. Actually, BFP has been applied to some limited applications[1][2], but efficient BFP implementations for other applications are not well known. In order to solve the problem of lacking the systematic implementation approach, hierarchical block-floating-point (H-BFP) arithmetic has been proposed[3][4]. The basic concept of H-BFP is to keep data on the memory in floating-point format, while processing data in fixed-point format. With H-BFP arithmetic, desired signal processing quality and reasonable hardware implementation can be obtained simultaneously like usual BFP. A good feature of H-BFP arithmetic is that the H-BFP implementation is directly derived from floating-point implementation. However, the application development approach presented in [4] is writing assembly code from scratch referring to the floating-point implementation of the target application. Manual translation of programs from high level language into assembly language is error prune and a time consuming task. A software development method which offers high productivity is required.

In this paper, an instruction-set processor supporting H-BFP arithmetic and its application development method are proposed. The proposed processor is designed based on the RISC architecture to enable compiler-based development. In the proposed method, H-BFP programs are implemented by modifying usual floating-point programs. The required modification is only to add special functions which are mapped into H-BFP instructions directly by compilers. Since the modification does not require any complicated program transformations, H-BFP program can be easily developed.

The rest of this paper is organized as follows. The H-BFP arithmetic is summarized in section 2, and an instruction-set processor with H-BFP arithmetic is presented in section 3. The software development approach for H-BFP processors is proposed in section 4. Experimental results are described in section 5. Finally, this paper is concluded in section 6.

2. H-BFP ARITHMETIC

This section describes the concept of H-BFP arithmetic[3] [4]. In the H-BFP, data elements are represented in floating-point format when they are on memories. During data processing, data elements are loaded from memories to the registers, then, they are converted from floating-point format into fixed-point format in the block-floating-point manner. Operations such as addition, multiplication and accumulation are performed on the data represented in fixed-point format. After the computations, results are converted into floating-point representation again, and stored into memories. Fig. 1 illustrates H-BFP arithmetic. As mentioned above, a set of data is located in data memory in floating-point representation. The floating-point to fixed-point converter is used to convert the data element. Data processing are performed on the fixed-point data path, and then the results are converted floating-point representation by the fixed-point to floating-point converter. The fixed-point to floating-point converter performs floating-point normalization. The results are stored to the data memory.

H-BFP has the advantage in terms of application productivity over conventional BFP arithmetic. In the development of conventional BFP based systems, design quality, such as signal quality, hardware cost and processing time, heavily depend on the implementation options. Application developers have to make effort to find a reasonable implementation. On the other hand, though H-BFP requires additional overhead in execution time due to data format conversions, high precision signal processing and low hardware cost are ensured.

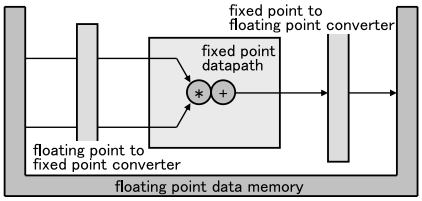

Fig. 1. H-BFP Arithmetic

3. H-BFP PROCESSOR ARCHITECTURE

An H-BFP processor has been designed based on the RISC architecture. Since the RISC architecture is simple, it is suitable for devices with limited hardware area. Another advantage of the RISC architecture is that high quality assembly code can be compiled from programs written in high level language.

Fig. 2 shows the proposed H-BFP processor architecture. The H-BFP data path is embedded in the conventional RISC pipeline at the third stage. The instruction set of the target processor is designed including both conventional RISC style instructions and H-BFP instructions. The target architecture is five stage pipelined; i.e., instruction fetch (IF), instruction decode (ID), execution (EX), memory access (ME), and write back (WB) stages. The H-BFP data path is equipped into

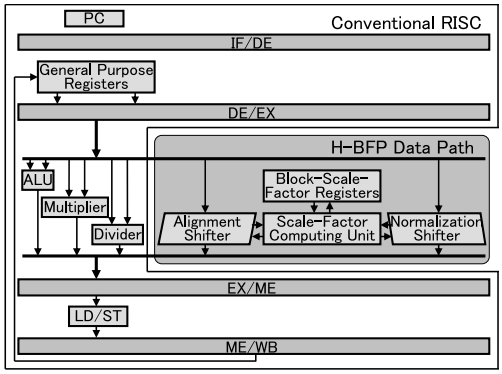

Fig. 2. H-BFP Processor Architecture

EX stage. There are 4 components in the H-BFP data path, *block-scale-factor registers*, *alignment shifter*, *normalization shifter*, and *scale-factor computation unit*. The block-scale-factor registers holds scale-factors of data blocks. The alignment shifter is used to convert floating-point to fixed-point numbers. The mantissa of a floating-point number is extracted and aligned by the shifter before computations. On the other hand, the normalization shifter is used to convert fixed-point numbers to floating-point numbers. The scale-factor computation unit manipulates block-scale-factors. Several kinds of block-scale-factor manipulations are required in the H-BFP arithmetic, such as block-scale-factor computations of data blocks, shift amount generation in float to fixed point conversions. The scale-factor computation unit is used for such scale-factor manipulations.

For H-BFP based processing, instructions which perform primitive operations of H-BFP are implemented. Floating-point to fixed-point and floating-point to fixed-point conversion instructions are implemented as register to register operations. Block-scale-factor manipulation instructions are also implemented as operations between block-scale-factor registers.

4. SOFTWARE DEVELOPMENT ENVIRONMENT

In this section, software development approach for H-BFP processor introduced.

4.1. Software Development Approach

Software for H-BFP processor is developed as follows. A program based on floating-point arithmetic of the target application is written in a high level programming language first of all. Then, the program is rewritten into the program based on H-BFP arithmetic. The differences of the computation model between the floating-point and H-BFP are that data are non-blocked or blocked, and data conversions after/before operations are not needed or needed. Hence, the software for H-BFP processor can be developed by adding H-BFP specific operations to the floating-point arithmetic based program. To enable the programmers to add such operations, a compiler

technique called compiler intrinsic is used. Compiler intrinsic functions are the functions in the high level programming language which are mapped to the specific instructions of the target processor. Using compiler intrinsic, H-BFP specific operations in H-BFP programs can be directly mapped to instructions of the H-BFP processor.

Fig.3 shows the development flow of conventional approaches and the proposed approach. In conventional fixed-point or BFP based software development, and H-BFP based software development presented in [4], floating-point arithmetic based program is developed first. Then, the target application is developed referring to the floating-point based program as a reference model. In the conventional fixed-point or BFP based software development flow, several implementations must be considered, and analysis of trade-off between signal processing quality and costs has to be performed. In the flow of [4], while the feature of H-BFP eases development, assembly programming is still needed. On the other hand, in the proposed approach, the target application program can be obtained easily because all have to do is program refinement by insertion of compiler intrinsic functions.

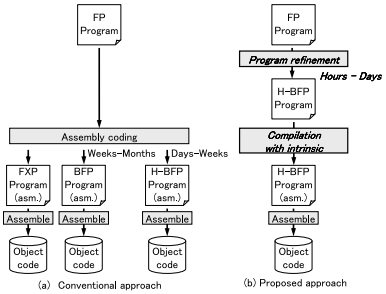

Fig. 3. Comparison of Application Development Flow

4.2. Program Refinement

Fig.4 shows a program refinement example of a floating-point program. There are three program fragments in Fig.4. The left program is the floating-point implementation for processors supporting floating-point arithmetic. The upper and lower programs at the right in Fig.4 are the H-BFP implementation and floating-point implementation for the H-BFP processor, respectively. The upper right program can be obtained by inserting scale-factor manipulations and data conversions in block-floating-point manner. On the other hand, the lower right program can be obtained by inserting H-BFP specific operations in floating-point manner. Compiler intrinsic functions, `sfcselect`, `tofix` , `toflt` , are appeared into their corresponding instructions for the H-BFP processor.

All these example programs compute the addition of two vectors. In the upper right program in Fig.4, the addition of two vectors is interpreted as addition of two vectors which belong to different data blocks. The variables `asfb` and `bsfb` in Fig.4 hold the block-scale-factors for data blocks `a` and `b` respectively. The `sfcselect` function computes the block-scale-factor which determines the fixed-point data format on the addition. In the loop body, `tofix` function performs floating-point to fixed-point conversion, `toflt` function performs fixed-point to floating-point conversion. By inserting scale-factor manipulation such that the H-BFP processor performs the manipulation before every addition as shown the lower right program, floating-point implementation on H-BFP processor can be obtained. The floating-point implementation takes more execution cycles than H-BFP implementation in the runtime. However, floating-point implementation achieves higher precision of arithmetic operations than H-BFP implementation.

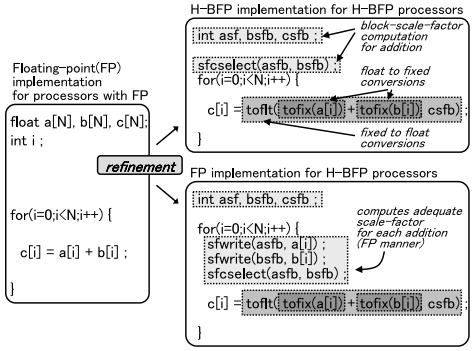

Fig. 4. Program Refinement for H-BFP processor

5. EXPERIMENTAL RESULTS

In this section, the experimental results are presented.

5.1. Signal Processing Quality

An HDL model of the H-BFP processor has been designed and the compiler with compiler intrinsics has been developed. The word length of the processor is set to 32 bits. The floating-point format on memory is composed of 8bits exponent and 16bits mantissa. The DSPstone benchmark[5] has been used for experiments. C programs implemented by floating-point arithmetic in DSPstone benchmark has been modified into H-BFP and floating-point implementations for the H-BFP processor. The HDL model of the H-BFP processor with the object code generated by the compiler has been simulated on an HDL simulator. The white noise has been used as the input of the programs.

The signal processing quality of H-BFP implementation has been evaluated using an signal-to-noise ratio measure which is defined as

$$SNR = 10log[\frac{1}{N}\frac{\sum_n DOUBLE(n)^2}{\sum_n \{DOUBLE(n) - TARGET(n)\}^2}] \quad (1)$$

where N is the number of outputs of the application, $DOUBLE(n)$ is the n th output of the double precision floating-point computation, and $TARGET(n)$ is the n-th output obtained by HDL simulation of the H-BFP processor, respectively.

Table1 shows the SNR of H-BFP and floating-point implementations for each program. The first column shows the

Table 2. Comparison of the number of insns. among different implementations, N : the size of vector, M : the width and height of matrix, T : the number of taps

	Pentium i386/gcc [# of insns]	H-BFP [# of insns]	FP [# of insns]
n_real_updates	26N+8	18N+13	27N+7
n_complex_updates	41N+5	45N+14	93N
complex_multiply	36N+4	31N+17	106N+7
lms	20N+17	29N+41	48N+5
convolution	17N+5	13N+24	27N+15
dot_product	17N+5	13N+23	27N+11
fir	7NT+3	12NT+13	24NT+19
matrix1	$8M^3+15M^2+21M+3$	$12M^3+9M^2+5M+5$	$26M^3+8M^2+5M$
matrix2	$19M^3+37M^2+7M+4$	$13M^3+30M^2+8M+16$	$27M^3+47M^2+8M+4$
mat1x3	$19M^2+12M+5$	$13M^2+16M+16$	$27M^2+13M+5$
fir2dim	$57M^2T+29M^2+6M$	$36M^2T+33M^2+9M+21$	$84M^2T+26M^2+6M+4$

Table 1. SNR of each programs run on the H-BFP Processor H-BFP : H-BFP impl., FP : floating-point impl.

	H-BFP [db]	FP [db]
n_real_updates	65.7	71.2
n_complex_updates	65.3	68.8
complex_multiply	67.9	71.5
lms	47.5	55.7
convolution	82.6	99.0
dot_product	79.1	87.3
fir	89.2	82.7
matrix1	68.3	67.0
matrix2	68.3	70.0
mat1x3	76.6	81.7
fir2dim	73.1	72.5

names of programs, the second and third columns shows the SNR of the H-BFP and floating-point implementations on the H-BFP processor, respectively. In Tab.1, SNRs of the H-BFP implementations ranged from 47.5 to 89.2. SNRs of the floating-point implementations score higher than those of the H-BFP implementations. It is confirmed that the H-BFP implementation on H-BFP processor achieves high precision of signals, and FP implementation on H-BFP processor can further improve precision.

5.2. Performance

To confirm the performance of the H-BFP processor/compiler, the size of assembly programs for H-BFP processor has been evaluated. Pentium i386/gcc was selected as a reference of floating-point system to compare the H-BFP processor/compiler. The type of instructions in assembly program includes arithmetic instructions, memory access instructions, jump and branch instructions, and floating-point instructions for Pentium/gcc or H-BFP specific instructions for H-BFP and FP. Without floating-point and H-BFP specific instructions, the number of instructions of Pentium/gcc and H-BFP assembly are almost same. Table 2 shows the number of instructions to process each program for Pentium i386/gcc and H-BFP and floating-point implementation of H-BFP processor/compiler. Comparing the H-BFP implementation with Pentium i386/gcc, the number of instructions of H-BFP implementation is smaller than that of Pentium i386/gcc in 7 cases. This result indicates the H-BFP processor/compiler can process applications as efficient as usual processor/compiler supporting floating-point arithmetic. However, the floating-point implementation of H-BFP processor, shown as FP, takes much instructions compared to Pentium i386/gcc. This is because not only data conversion operations but also scale-factor manipulations are executed in the floating-point implementation for each operation.

5.3. Hardware Evaluation

The hardware area of the H-BFP processor was estimated. The HDL model of the H-BFP processor was synthesized using a 0.14μm process. The total area of the H-BFP processor is about 67K gates with the maximum frequency at about 83.3MHz. The total area of the components in the H-BFP datapath is about 2.5K gates. The maximum delay of the H-BFP datapath is about 9.96ns, which is shorter than the delay of the critical path of the entire H-BFP processor, 12.0ns. It was confirmed that the hardware for H-BFP is very small and reasonable.

6. CONCLUSION

In this paper, a processor supporting hierarchical block-floating-point arithmetic and software development method for the processor are proposed. In experiments, some applications has been implemented and simulated on the H-BFP processor. It is confirmed that the H-BFP processor can achieve high signal quality and low hardware cost. Using the proposed method, signal processing applications can be easily developed.

The future work is to enhance architecture and instruction-set of the H-BFP processor, and automatic utilization of H-BFP instructions by compilers.

7. REFERENCES

[1] K. Ralev and P. Bauer, "Implementation Options for Block Floating Point Digital Filters," in *Proc. of ICASSP-97*, 1997, pp. 2197–2200.

[2] A. Mitra and M. Chakraborty, "The NLMS Algorithm in Block Floating Point Format," *IEEE Signal Processing Letters*, pp. 301–304, 2004.

[3] S. Kobayashi and G. Fettweis, "A New Approach for Block-Floating-Point Arithmetic," in *Proc. of ICASSP-99*, 1999, vol. 4, pp. 2009–2012.

[4] S. Kobayashi, I. Kozuka, and T. Kino, "Rapid Application Software Development on a Block-Floating-Point DSP," in *Proc. 2003 International Signal Processing Conference*, 2003.

[5] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology," in *Proc. of ICSPAT'94*, 1994.