Block Floating Point Interval ALU for Digital Signal Processing*

Sandeep Hattangady, William Edmonson, Winser Alexander
Department of Electrical and Computer Engineering,
North Carolina State University, Raleigh, USA
hattangady@gmail.com,wwedmons@ncsu.edu,winser@ncsu.edu

Abstract

Numerical analysis on real numbers is performed using either point-wise arithmetic or interval arithmetic. Today, interval analysis is a mature discipline and finds use not only in error analysis but also control applications such as robotics. The interval arithmetic hardware architecture proposed in the work [W. Edmonson, R. Gupte, J. Gianchandani, S. Ocloo, W. Alexander, Interval arithmetic logic unit for signal processing and control applications, Workshop on *Reliable Engineering Computing*, Savannah, GA, USA, 2006] for Digital Signal Processors (DSP) is prone to unreliability owing to overflow errors resulting from the small dynamic range offered by fixed point computations. In this paper, we present a solution to this problem in the form of a fixed point interval ALU that utilizes the concept of Block Floating Point (BFP) presented in [K. Kalliojarvi, J. Astola, [Roundoff errors in block-floating-point systems, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 44, pp. 783–790, 1996] to attain a higher dynamic range for interval as well as point-wise computations. We add block floating point support to the ALU in the form of special instructions such as Exponent Detection and Normalization that aid the change of the dynamic range of the input. We also provide additional hardware to perform Conditional Block Floating-point Scaling (CBFS) on the output endpoints of the interval result. We explore the design space across varying pipeline depths for best throughput which is characterized as the highest number of output samples processed per second. Our results show that the four-stage highly-pipelined architecture provides the highest throughput of about 86.1 Msamples/sec for interval operations.

Keywords: Block floating point, interval arithmetic, arithmetic logic unit.
AMS subject classifications: 97P60, 65G30, 65G40

*Submitted: January 27, 2009; Accepted: February 9, 2010.

1 Introduction

Digital signal processing and control applications use algorithms based on interval arithmetic to address a wide range of complex problems including solving systems of nonlinear equations, determining eigenvalues and eigenvectors of matrices, finding roots of functions and performing global optimization [1][2][3]. The demands of high computational performance for these applications can be met by Digital Signal Processors (DSP) which take advantage of specialized hardware architectural features such as fast multiply-accumulate instructions, multiple-access memory, parallel operations, specialized program control for interrupt handling and I/O, and fast and efficient access to peripherals. DSPs based on fixed point implementations are cheaper and less power hungry than their floating point counterparts given comparable speeds. However, fixed point implementations face the disadvantage of limited dynamic range and this can cause wrap around in 2's complement arithmetic leading to poor fidelity for signal processing applications. Therefore, interval computations cease to be reliable for the architecture presented in the work of [4]. Overflow is not much of an issue for floating point implementations because they have higher dynamic range [5].

In this paper, we provide a solution to mitigate the effect of overflow errors that beset the dedicated fixed point interval ALU [4]. We propose an interval ALU architecture that uses Block Floating Point (BFP) arithmetic to achieve higher dynamic range. The skeleton of our architecture is derived from the work of [4]. Since the underlying fixed point architecture is still prone to overflow errors, we handle it through the technique of Conditional Block Floating-point Scaling (CBFS). We extend BFP support to the ALU by incorporating instructions that perform Exponent Detection and Normalization. We also incorporate modifications in this architecture so that the ALU can also perform point-wise operations. We create a design space comprised of pipelined Block Floating Point Interval ALU (BFPIALU) architectures and estimate average throughput for each design.

This paper is organized as follows. Section II presents published work that provides the basis and motivation for this paper. Section III extends the concept of BFP arithmetic to interval arithmetic using CBFS for overflow handling. Section IV describes the basic hardware architecture of the non-pipelined BFPIALU and extends pipelining to this architecture for higher throughput. Section V presents the performance analysis for these designs and Section VI concludes the paper.

2 Related Work

Floating point interval hardware implementations have been carried out previously [6] [7] [8] [9] as a solution to address the poor execution rate of interval operations in software. However, there is only one dedicated fixed point interval ALU for DSP and control applications that has been designed and tested [4]. This design has recorded a competitive throughput on the order of 56 MIOPS (Millions of Interval Operations per Second) for the non-pipelined design and about 307 MIOPS for a 7-stage pipelined design [4]. A shortcoming of the design is that overflow errors lead to unreliable interval arithmetic and this problem is addressed in this paper.

We explore BFP arithmetic for the fixed point interval ALU to attain a dynamic range higher than that allowed by conventional fixed point arithmetic. BFP arithmetic has been successfully applied to point-wise computation based applications such as digital filters [10] and the Fast Fourier Transform [5] [11]. BFP arithmetic support

for point-wise computations in most DSPs is typically provided in the form of *Exponent Detection* and *Normalization* instructions [5]. Commercial fixed-point DSPs such as Analog Devices ASDP-21xx, Texas Instruments TMS320C54x [5], SGS-Thomson D950-CORE, Zoran ZR3800x, DSP Group OakDSPCore and uPD7701x provide single cycle Exponent Detection. However, DSPs from the AT&T DSP16xx family (other than DSP1602 and DSP1605) are the only ones that provide single cycle Normalize instructions. Other DSPs such as the Texas Instruments TMS320C2x, TMS320C5x, the DSP Group PineDSPCore, the Motorola DSP5600x and DSP561xx provide iterative normalization instructions where an n-bit number takes n-cycles to normalize [5]. However, none of these DSPs provide specialized BFP support for interval arithmetic. Overflow will be handled through the CBFS scheme [12] [13].

3 Interval Block Floating Point arithmetic

We consider a fixed point interval arithmetic system for the BFPIALU, presented in Section V, based on setting the interval endpoints to finite values, that abides by the set of criteria mentioned in the work of [14]. The criteria for evaluation of hardware is: correctness, totality, closedness, optimality, and efficiency. This section discusses BFP arithmetic with interval numbers and the CBFS scheme for handling overflow.

3.1 BFP Arithmetic

The BFP algorithm is based on the Automatic Gain Control (AGC) concept. Block AGC only scales the value of the data at the input stage of the data processing, thus only adjusting the input signal power. The block floating point algorithm takes it a step further by tracking the signal strength from stage to stage to provide a more comprehensive scaling strategy and extended dynamic range [15].

A block of N numbers has a joint scaling factor of γ corresponding to the largest magnitude of the data in the block for BFP representation. We extend the definition of BFP representation presented in [12] for point-wise data, to interval data. If $[x_{Li},\ x_{Ui}]$ represents the i^{th} interval data sample and γ represents the interval block exponent, then the BFP representation is denoted as

$$[[x_{L1}, x_{U1}], [x_{L2}, x_{U2}], \ldots, [x_{LN}, x_{UN}]] = [[\hat{x}_{L1}, \hat{x}_{U1}], [\hat{x}_{L2}, \hat{x}_{U2}], \ldots, [\hat{x}_{LN}, \hat{x}_{UN}]] \cdot 2^{\gamma},$$

where $[\hat{x}_{Li}, \hat{x}_{Ui}] = [x_{Li}, x_{Ui}] \cdot 2^{-\gamma}$. The block exponent γ is defined by

$$\gamma = \lfloor \log_2 M \rfloor + 1 + S$$

where M = max($|x_{L1}|$, $|x_{U1}|$, ...,$|x_{LN}|$, $|x_{UN}|$) and $|\hat{x}_{Li}| \in$ [0,1]; $|\hat{x}_{Ui}| \in$ [0,1]. The integer S signifies a constant integer scaling term for the block exponent and will affect both endpoints in a similar way. The same exponent is applied to both endpoints of each interval data since we intend to accommodate overflow errors in fixed point hardware. Furthermore, the complexity of hardware will approach that of a floating point implementation if different exponents are used for the lower and upper endpoints of the interval data.

A more intuitive approach to compute the value of γ for the block is to assign it with the negated count of the leading zeros for the sample with the largest magnitude in a data block. Data with small magnitudes in two's complement arithmetic normally

fit to the machine register size by size extension. Therefore, the data with the least number of leading sign bits has the largest magnitude and we exploit this fact in computing the value of γ [12]. Once the block of interval data is normalized, we may then perform the interval operations on it.

3.2 CBFS

This technique is based upon the idea of *correcting* overflow errors. The output block exponent is determined *a posteriori* without using the constant integer scaling term 'S' during normalization. After normalization, the data samples are brought into the fixed point hardware for computations. If no overflow occurs, then the output block exponent is kept the same as the normalized input block exponent. However, if overflow occurs, then a set of corrective actions are taken. These are listed below:

1. The hardware block scales the overflowed output down by a factor of 2. This is performed by right shifting the result of the operation in fixed point.
2. The output block exponent is incremented and the result is stored.
3. If the computations are iterative in nature and an intermediate overflow occurs, then all inputs from that point onwards are scaled down by an additional factor of 2. This process is repeated at each instance of overflow.

CBFS implementations do not scale the input data unnecessarily during normalization and set the value of parameter S to 0. The operation is first performed and the output is scaled down by a factor of two only in the event of overflow. CBFS implementations that run long iterative operations, such as computation of dot products, increment the output block exponent only if overflow occurred. We do not consider the *saturation* scheme for the interval architecture since it could lead to underestimation of the output interval bounds and hence not meet the criterion of *correctness*. The design of the BFPIALU, proposed in this work implements the CBFS technique.

4 Hardware Architecture

The hardware architecture for the ALU is shown in Figure 1. It comprises of the Flag Generator module, the Lower Bound module, the Upper Bound module, the Scale Synchronizer module and the input scaling modules Scale_L and Scale_U. We choose the Q0.15 data format for fixed point data representation to obtain higher precision. We provide the ability to handle overflow errors due to the small dynamic range obtained as a result.

The ALU takes two intervals X and Y with corresponding endpoints X_L, X_U, Y_L and Y_U as inputs. The ALU has two modes of operation, namely the *interval* mode and *pointwise* mode for interval and point arithmetic operations respectively. The choice of the mode of operation is dictated by the *mode* signal. Asserting this signal high causes the ALU to function in the *interval* mode while asserting it low causes the ALU to operate in the *pointwise* mode of operation. In the *interval* mode of operation, both the Lower Bound and Upper Bound modules operate on the same command. However, in the *pointwise* mode of operation, both modules can execute different commands independently. This explains the presence of two copies of command input *cmd* and data / control lines such as *maxexp*, *mac* and *permitscaling*. The signals *macexp1* and *maxexp2* are input lines that convey the amount of shifting to be performed on the input data. This input is important while normalizing a data block. Asserting the *mac*

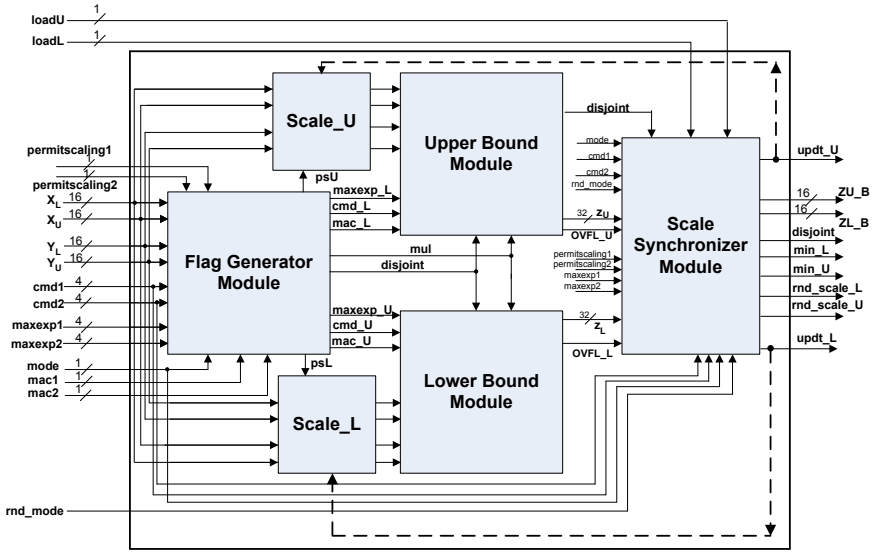

Figure 1: Hardware Architecture

signals high results in an accumulation of products as long as this signal is asserted high. Signal *permitscaling* indicates an iterative operation when asserted high. For situations that demand iterative operations to be resumed midway, signals *loadL* and *loadU* can be asserted to load the value of the increment in the output block exponent. The input *rnd_mode* is used to make the choice of a rounding scheme in the Upper and Lower Bound modules when the interval ALU is in the *point̄wise* mode of operation. Therefore, both modules can function as two independent ALUs that can perform *point̄wise* operations in parallel.

Outputs ZL_B and ZU_B signify the endpoints of the interval output interval. The output *min_L* and *min_U* indicate the least exponent detected value among all samples of a data block. These are updated alongside the process of Exponent Detection in the BFPIALU. The outputs *updt_L* and *updt_U* indicate the increment in the output block exponent compared to the input data block exponent. The signals *rnd_scale_L* and *rnd_scale_U* are used to indicate a special case of scaling performed to account for overflow due to rounding to $+\infty$ in either the Lower Bound or the Upper Bound modules.

The BFPIALU is capable of performing the operations mentioned in Table 1. While commands 0-7 have been implemented in [4], commands 8-F represent the extended set of operations in this ALU. The logic blocks that are used to perform these extended operations are described later in the chapter. Both commands *cmd1* and *cmd2* can assume these values at any point of time. The MAC (multiply-accumulate) operation is performed as part of multiplication. The corresponding *mac* signal is asserted high along with the *multiply* command to perform this operation. We provide a detailed description of the individual modules comprising the BFPIALU architecture shown in Figure 1.

Command	*Description*
0	ADDITION
1	SUBTRACTION
2	MULTIPLICATION / MAC
3	DIVISION
4	UNION
5	INTERSECTION
6	WIDTH
7	MIDPOINT
8	MIN
9	MAX
A	OR
B	AND
C	XOR
D	EXPONENT DETECTION
E	NORMALIZATION
F	SIGNED LEFT SHIFT

Table 1: Command Set for the ALU

4.1 Flag Generator

The Flag Generator identifies the appropriate case of interval multiplication to be performed and flags disjoint input intervals. It also handles the distribution of appropriate control signals such as *cmd*, *maxexp*, *permitscaling* to the Lower and Upper Bound modules depending on the mode of operation.

4.2 Lower and Upper Bound Modules

Both the Lower and Upper Bound modules are largely similar in structure. The operations in each of these modules are performed in parallel with one logic path dedicated to each operation. Changes in these modules as compared to the work of [4] include the addition of block floating point operations and the elimination of the *special multiplication block* and the *next* signal for set operations.

The inputs to the Lower Bound module are obtained from the Scale_L module and are denoted by X_L, X_U, Y_L and Y_U for convenience. Each of these inputs represent scaled values of the original interval inputs.

We next describe the logic used to realize the operations of Exponent Detection and Normalization.

Exponent Detection and Normalization: The operation of Exponent Detection involves identifying the number of redundant sign bits in a given data element. For a given data element, it is done by XORing successive bits and then passing the result through an array of priority encoders. The output is an integer corresponding to the number of redundant sign bits in the input data element depending on where the first combination of 01 or 10 occurs in the input. Exponent detection for interval data is performed by identifying the minimum number of redundant sign bits in the lower and upper end point of the applied interval. The result (γ) is updated in both the *min_L* and *min_U* output registers. Once an interval data block is traversed through, the

value of γ is applied to the *maxexp1* or *maxexp2* and the entire interval data block is left shifted. This normalizes the interval data block.

4.3 Scale Synchronizer Module

Overflow can occur in either the Lower Bound module, the Upper Bound module, or in both. In order to obtain a reliable result, the overflow detection circuitry, housed within each of these modules, scales down the output of the operation by a factor of 2. It is advantageous to set both output interval endpoints to the same scaling level in hardware because this would enable both endpoints to bear the same output exponent value. Performing this step in hardware avoids the time penalty associated with checking the status of scaling of individual endpoints for each output interval while normalizing the output interval data block for the next stage of block processing. The Scale Synchronizer block takes two 32-bit inputs, namely Z_L and Z_U from the outputs of the Lower Bound and Upper Bound modules respectively. It rounds these 32-bit values to 16-bit outputs with the choice of the appropriate rounding scheme. It synchronizes the scaling on Z_L and Z_U and updates registers *updt_L* and *updt_U* to store the increment in output block exponent, depending upon the status of overflow signals from the Lower Bound and Upper Bound modules. It also stores the minimum exponent detected in a data block during Exponent Detection, so that Left shifting can follow immediately for Block Normalization.

Special case of Overflow

The outward rounding scheme for intervals is retained from the work of [4]. Therefore, the 32-bit result from the Lower Bound module is truncated to 16-bits while the output of the Upper Bound module is rounded to $+\infty$. Rounding to $+\infty$ entails the addition of the OR-ed result of the lower-precision bits, to be discarded, to the rest of the bits. However, this can lead to unintentional overflow when the 32-bit result is of the form 0x7FFFXXXX hex where XXXX is non-zero. This is referred to as the *Special case of Overflow*. This overflow results in 8000 hex resulting in an incorrect result. Such a situation can occur in the Upper Bound result in either the *interval* mode or *pointwise* mode and in the Lower Bound result in the *pointwise* mode of operation. This special case is addressed by flags *rnd_scale_L* and *rnd_scale_U* which are asserted high by the Scale Synchronizer module when such a situation is detected. Corrective action is taken by putting out a result of 4000 (hex) and setting the relevant flag to indicate this situation. This also results in γ incrementing by 1.

Synchronizing the outputs

The synchronization of the interval output endpoints and the value of the updated registers *updt_L* and *updt_U* is illustrated in Table 2. Signal *psL* represents *permitscaling1*, *OVFL_L* and *OVFL_U* indicate the status of overflow in the Lower and Upper Bound modules while *SplRnd* indicates whether the special case of rounding occurred. Both output endpoints share common block exponent for an interval operation and the same value is loaded in the output registers *updt_L* and *updt_U*. Updating registers *updt_L* and *updt_U* in the event of overflow for iterative computations involves incrementing their previous values. For non-iterative operations, the output block exponent increment is simply set to 1 when overflow occurs. It is important to note that the input is scaled by a factor equal to the output block exponent increments. The same principle is extended for point-wise operations by updating the output block exponent registers, *updt_L* and *updt_U*, independent of each other. Furthermore, this

psL	*OVFL_L*	*OVFL_U*	*SplRnd*	*ZL_B*	*updt_L*	*ZU_Bhex*	*updt_U*
0	0	0	0	zl	0	zu	0
0	0	0	1	zl/2	1	4000	1
0	0	1	0	zl/2	1	zu	1
0	0	1	1	zl/4	2	4000	2
0	1	0	0	zl	1	zu/2	1
0	1	0	1	zl	1	4000	1
0	1	1	0	zl	1	zu	1
0	1	1	1	zl/2	2	4000	2
1	0	0	0	zl	updt_L	zu	updt_L
1	0	0	1	zl/2	updt_L+1	4000	updt_L+1
1	0	1	0	zl/2	updt_L+1	zu	updt_L+1
1	0	1	1	zl/4	updt_L+2	4000	updt_L+2
1	1	0	0	zl	updt_L+1	zu/2	updt_L+1
1	1	0	1	zl	updt_L+1	4000	updt_L+1
1	1	1	0	zl	updt_L+1	zu	updt_L+1
1	1	1	1	zl/2	updt_L+2	4000	updt_L+2

Table 2: Scale Synchronization for Interval Operations

module houses the logic to identify the value of γ by identifying the least number of leading sign bits in the given block of data samples. The output registers *min_L* and *min_U* store the value of γ identified for data samples applied to the Lower and Upper Bound modules respectively. Interval operations result in a common value for both the registers while point-wise operations can result in distinct values. The operation of Exponent Detection begins with the value of γ being set to a maximum value of 1111_2 and the value is replaced successively with smaller values as data samples are passed through the ALU. The values of *min_L* and *min_U* registers at the end of this procedure correspond to the value of the exponent for the entire data block.

4.4 Scaling Modules

Overflow in the intermediate stages of iterative computations must be handled by scaling the inputs down by a factor equal to the output block exponent. Thus operation is performed by the modules, *Scale_L* and *Scale_U*, which scale the inputs to the Lower and Upper Bound modules respectively. Both the modules contain right-shifted versions of the input subjected to appropriate rounding to ensure that no underestimation of the input is performed.

4.5 Pipelined Architecture

The non-pipelined architecture described above presents a critical path upon synthesis, starting in the Flag Generator module, passing through one of the multipliers in the Upper Bound module and terminating in the Scale Synchronizer module. The multiplier contributes a significant cloud of logic towards the critical path. Therefore, the use of pipelined multipliers provides scope for reducing the logic depth of this path resulting in a faster design. We refer to the pipelined designs obtained in this manner as *Normally pipelined designs*.

Using only pipelined multipliers to achieve designs with four stages of pipelining or higher does not yield significant timing gain. Hence, we insert a register at the

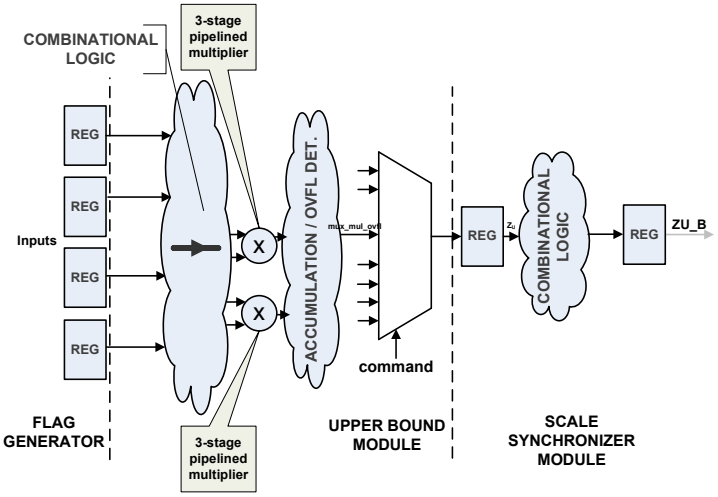

Figure 2: Critical Path for the Highly-pipelined design

boundary of the Lower and Upper Bound modules with the Scale Synchronizer module in the three-stage normally-pipelined design to reduce the logic depth of the critical path. This results in a four-stage pipelined design. Since this is the fastest design obtained upon exploring various ways of pipelining the design, we refer to it as the *Highly pipelined design*. The critical path for this design is shown in Figure 2.

5 Results

5.1 Area, Timing and Power Results

The various architectures of the ALU are synthesized using the 0.18μm standard cell library from Oaklahoma State University [16]. Synopsys Design Compiler [17] was used to perform the area and timing analysis. Synopsys Prime Power [17] was used to perform power analysis using the netlist corresponding to each pipelined architecture and simulating it 1000 random input interval vectors. The area, timing and average power dissipation values obtained for the various architectures are presented in Table 3.

Table 3: Area and Timing Report for the Synthesized Architectures

Pipeline Stages	*Least Timing* (ns)	*Area* (μm^2)	*Power* (W)
Nonpipelined	10.33	473703	0.04918
2-stage Normally Pipelined	7.08	483855	0.06978
3-stage Normally Pipelined	5.10	506727	0.09033
4-stage Normally Pipelined	5.04	557637	0.1032
4-stage Highly Pipelined	3.87	527184	0.1247

5.2 Evaluating Throughput for the Highly-Pipelined Architecture

Throughput for an interval ALU is ideally defined as the number of interval operations performed per second [4]. However, throughput for a BFP implementation depends upon many factors such as the number of cycles per output sample, the time it takes to normalize a data block and the effect of overflow. In this section, we redefine throughput and evaluate it for interval operations based on a chosen scheme of computations performed in the BFPIALU.

5.2.1 Probability of Overflow

Each overflow in the ALU is dealt with by scaling the inputs down by a factor of 2. This corresponds to doubling the available dynamic range in the ALU and clearly indicates a reduced probability of the occurrence of the next overflow. A Monte-Carlo simulation with the accumulation of products of uniformly distributed inputs to the ALU was performed which confirmed this trend. This fact allows us to ignore any computation-cycle penalties associated with overflow in pipelined designs. In general, N-stage pipelined designs present an overhead of at least N clock cycles upon the occurrence of each overflow owing to the propagation of the output block exponent to the input Scaling modules.

5.2.2 Evaluation of Throughput

We define throughput (R) for the BFPIALU in terms of *Output Samples per Second*.

$$R = \frac{Number\ of\ Samples\ Processed}{Time\ to\ process\ the\ Samples}$$

The time to perform operations that do not involve feedback in their paths or need not be performed iteratively can be looked up directly from Table 3 as the reciprocal of the clock period. We, however, focus on the throughput obtained with the MAC operation, since it is important for DSP applications. We assume, for our analysis, a data block comprised of N interval data samples to evaluate the throughput for interval operations. We also assume that the architecture is to be evaluated for throughput contain k pipeline stages and that the computation for each block results in p number of overflows. For normally pipelined designs, the throughput is obtained [18] as

$$R = \frac{N}{[3N + 2(k-1) + p(k+2)] \cdot t} \tag{1}$$

We next subject the MAC operation in the highly-pipelined design to a similar analysis. The throughput in this case is obtained [18] as

$$R = \frac{N}{[3N + p(k+3) + 2(k-1) + 1] \cdot t} \tag{2}$$

In the limiting case of $N \to \infty$, $n \gg k$ for both (1) and (2), the throughput evaluates to

$$R = \frac{1}{3 \cdot t}$$

Clocked at 3.87ns, the highly pipelined design has the highest throughput of 86.1 Msamples/sec. This represents 166% improvement over the throughput of the non-pipelined design (k=1), which was clocked at 10.33ns recording a mere 32.2 Msamples/sec, as well all other normally-pipelined designs. Thus, the highly pipelined architecture provides the highest throughput among all the pipelined designs of the ALU for the chosen scheme of computation.

Under ideal circumstances, where no structural hazard [18] occurs, the throughput of the designs for point-wise operations may be stated to be exactly twice that of the interval operations. Therefore the point-wise operations ideally record a throughput of 172.2M samples/second for MAC and non-MAC operations.

6 Conclusion

We have presented the architecture for a BFPIALU that handles overflow using the CBFS scheme in this paper. We apply this scheme to operations susceptible to overflow such as addition, subtraction and special instructions specific to DSP and control applications such as multiply-accumulate. We have presented the basic architecture first and then explored the design space comprised of designs with varying depths of pipelining to obtain better performance. We identify the highly-pipelined architecture, distinguished with a three-stage pipelined multiplier and a delay element at the boundary of the Lower and Upper Bound modules with the Scale Synchronizer module, as the design that yields the highest throughput, measured as the number of samples processed per second. This design records a throughput of 86.1 Msamples/sec in comparison to the basic design which records a throughput of only 32.2 Msamples/sec with a latency of one clock cycle.

References

[1] S. Ocloo, W. Edmonson, An Interval-based Algorithm for Adaptive IIR Filters, Fortieth Asilomar Conference on *Signals, Systems and Computers (ACSSC)*, pp. 258–262, 2006.

[2] E. Hansen, G. W. Walster, *Global Optimization using Interval Analysis*, Marcel Dekker, Inc. and Sun Microsystems, Inc., 2004.

[3] R. Shettar, R. M. Banakar, P. S. V. Nataraj, Design and implementation of interval arithmetic algorithms, Proc. International Conference on *Industrial and Information Systems*, Peradeniya, Sri Lanka, pp. 328–331, 2006.

[4] R. Gupte, W. Edmonson, S. Ocloo, W. Alexander, Pipelined ALU for signal processing to implement interval arithmetic, The IEEE 2006 Workshop on *Signal Processing Systems* (SiPS06), Banff, AB, Canada, pp. 95–100, 2006.

[5] P. Lapsley, J. Bier, A. Shoham, E. A. Lee, *DSP Processor Fundamentals - Architectures and Features*. Institute of Electrical and Electronics Engineers Inc., 1997.

[6] A. Amaricai, M. Vladutiu, L. Prodan, M. Udrescu, B. Oana, Design of addition and multiplication units for high performance interval arithmetic processor, Proc. 10th IEEE Workshop on *Design and Diagnostics of Electronic Circuits and Systems*, Krakow, Poland, pp. 1–4, 2007.

[7] M. J. Schulte, J. E. E. Swartzlander, A family of variable precision interval arithmetic processors, *IEEE Transactions on Computers*, vol. 49, pp. 387–398, 2000.

[8] J. E. Stine, M. J. Schulte, A combined interval and floating-point multiplier, 8th Great Lakes Symposium on *VLSI*, pp. 208–213, 1998.

[9] A. Akkas, A combined interval and floating-point comparator/selector, Proc. IEEE 13th International Conference on *Application-Specific Systems, Architectures and Processors*, San Jose, USA, pp. 208–217, 2002.

[10] A. Oppenheim, Realization of digital filters using block-floating-point arithmetic, *IEEE Transactions on Audio and Electroaccoustics*, vol. 18, pp. 130–136, 1970.

[11] S. Kobayashi, S. Y. Lee, T. Kino, I. Kozuka, and T. Tokui, Audio application implementations on a block-floating-point dsp, *IEEE Workshop on Signal Processing Systems*, pp. 51–56, 2002.

[12] K. Kalliojarvi, J. Astola, Roundoff errors in block-floating-point systems, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 44, pp. 783–790, 1996.

[13] A. C. Erickson, B. S. Fagin, Calculating the FHT in hardware, *IEEE Transactions on Signal Processing*, vol. 40, 1992.

[14] V. Emden, T. Hickey, Q. Ju, Interval arithmetic: From principles to implementation, *Massachusetts Journal of the ACM*, vol. 48, pp. 1038–1068, 2001.

[15] A. Chhabra, R. Iyer, A block floating point implementation on the tms320c54x dsp, Technical report, Texas Instruments, December 1999. Application report SPRA610.

[16] The Oaklahoma State University System on Chip (SOC) Design Flows, http://vcag.ecen.okstate.edu.

[17] `http://www.synopsys.com`

[18] `http://www.lib.ncsu.edu/theses/available/etd-09242007-104624/unrestricted/etd.pdf`