ABSTRACT

HATTANGADY, SANDEEP K. Development of a Block Floating Point Interval ALU for DSP and Control Applications. (Under the direction of Professor Willam W. Edmonson).

With the advent of interval arithmetic, numerical analysis on real numbers has come to be classified into *theoretical analysis* or analysis based on point-wise arithmetic, and *interval analysis* or analysis based on interval arithmetic. With computational reliability gaining importance, interval analysis has been proposed as a technique to provide a certificate of reliability to the computations. However, software implementations for interval arithmetic show poor execution rates. Therefore, computationally intense applications in digital signal processing and control systems resort to fixed-point hardware implementations, which provide better solutions to these problems with high throughput. However, fixed point architectures are susceptible to overflow errors leading to unreliable results, which cannot be tolerated with interval operations in particular.

This work develops a Block Floating Point Interval ALU (BFPIALU) to attain reliable interval arithmetic on fixed point architectures. BFP support is provided through the ability to perform special BFP operations such as Exponent Detection and Normalization in its command set. Overflow is handled by a need-based scaling technique known as Conditional Block Floating Point Scaling (CBFS) technique. The ability to perform point-wise computations is also included by incorporating modifications in the interval architecture that allow it to function as two parallel ALU units for such computations.

This work models throughput for the pipelined BFPIALU architectures in terms of the clock rate, the number of pipeline stages and the number of overflows. It presents a four-stage pipelined architecture that can provide a throughput of 86.1 M samples per second and perform upto 258.4 million interval operations per second. The architecture can also perform 516.8 million point-wise operations per second.

Development of a Block Floating Point Interval ALU for DSP and Control Applications

by

Sandeep K. Hattangady

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Electrical and Computer Engineering

Raleigh, North Carolina

2007

Approved By:

Dr. Winser E. Alexander	Dr. William Rhett Davis

Dr. William W. Edmonson
Chair of Advisory Committee

Dedication

To

Amma, Pappa and Akka

Biography

Sandeep Hattangady was born on November 2, 1982 in Honavar, India. He received his Bachelor of Engineering (B.E.) degree in Electronics and Communication from MS Ramaiah Institute of Technology (Visveswaraiah Technological University) in June 2004. He worked at Infosys Technologies for an year as a software engineer and later joined the Graduate program at NC State University in Electrical and Computer Engineering in August 2005.

He has been working with the HiPer DSP Group under Dr William Edmonson since Spring 2006 in the field of hardware development for interval arithmetic and digital signal processing. He has also been part of a collaborative effort on a space based project between the NC State University, the University of Florida, Gainsville and the Defense Advanced Research Projects Agency (DARPA). He is a student member of the Institute of Electrical and Electronics Engineers (IEEE).

He has a keen interest in music and is part of 'BombayJukebox', a music band in Raleigh. He is also a professional percussionist for Indian classical music and plays the Tabla and the synthesizer. His research interests include computer arithmetic, ASIC design and verification and music cognition.

Acknowledgements

I thank my parents, Radhika and Krishnanand Hattangady, and sister, Deepa Karnad, for the strength, courage and moral support they provided me with not just during the course of this thesis but also during my stay so far away from home. I am highly indebted to Krishna Joshi, Sampada Joshi and their family for their love, concern and support for me during this time. Without their support, this thesis would not have seen the light of the day.

I sincerely thank Dr. William Edmonson, my advisor, for shaping my perspective on research, life and teaching me the importance of expressing thoughts through writing. I have learned to answer the five W's in everything I do, from him. He has given me immense opportunities and freedom to explore my own ways of solving the problem and has guided me to the solution I have today. His constant flow of unconventional ideas has amazed me, and inspired me to research and analyze more. I am sincerely thankful to him for his efforts in making my thesis readable.

I thank Dr. Winser Alexander and Dr. Rhett Davis for their valuable inputs on my work during the course of my thesis. Their guidance and suggestions have played a very important role in shaping this work.

I would like to thank all members of the HiPer DSP group at NC State University for their support. In particular, I would like to express my gratitude to Ramsey Hourani, Ravi Jenkal, Senanu Ocloo, Young Soo Kim and Cranos Williams for all the help they gave me and for being there for me when I needed them the most. Each one of them has provided me with constructive suggestions throughout the course of my Masters and shaped my opinions on many aspects of academics and life. I am indebted to Ravi, Mithun Acharya and Yathiraj Udupi, forever, for the gift of food and shelter they provided me with when I first landed on western soil two years ago.

I am highly grateful to Kaushal Mishra, Sujit, Maitrik, S. Raghunandan Sharma, Partha, Mohan, Adwait, Jitu, members of the 'BombayJukebox' and all near and dear ones for their unconditional support and help throughout.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern numerical analysis with real numbers includes the techniques of *interval* analysis and *theoretical* analysis [1]. Interval analysis performs arithmetic on *ranges* of real numbers known as intervals, whereas theoretical analysis performs arithmetic on *exact* real numbers. Today, interval analysis is a mature discipline and finds use in not only digital signal processing applications such as fuzzy adaptive filtering [2] and error analysis [1] but also control applications such as decision systems [3]. Knowledge-based systems employ intervals to model imprecise quantities such as knowledge [4]. Many interval-based algorithms have been developed to address more and more complex problems such as solving systems of nonlinear equations, determining eigenvalues and eigenvectors of matrices, finding roots of functions, and performing global optimization [5].

Recognizing the growing importance of interval-based algorithms, software packages such as the Sun Forte Fortran 95 compiler, the GNU Fortran compiler, the Sun C / C++ compiler, Frink programming language, Boost C++ package and many others provide support for interval arithmetic. The complete list of software that support interval arithmetic is available at [6]. However, software implementations of the interval-based algorithms have fallen well short of expectations in terms of per-

formance [7]. The cause for this is attributed to the overhead due to function calls, memory management issues, error and range checking, changing rounding modes, exception handling and many others [8]. Furthermore, checking the sign of the input interval endpoints for interval multiplication leads to a set of conditional statements. This, in turn, could lead to frequent flushing of pipelines in a processor. These issues are computationally costly and can be mitigated by providing hardware support for interval operations.

Gupte, R. et.al. [9] implemented a fixed point Arithmetic Logic Unit (ALU) dedicated to interval computations for digital signal processing and control applications to address the problem of slow program execution. While the interval ALU is competitive in its throughput and power consumption, it is prone to *overflow errors* owing to bit growth beyond the limits of the fixed point numeric representation. Overflow can occur when frequently used operations such as multiply-accumulate are performed successively a large number of times in this implementation. Interval arithmetic can cease to be reliable and this defeats the main purpose of using it. Interval arithmetic aims to provide reliable bounds on the results of point-wise evaluations, thereby providing a certificate of reliability to such computations.

This work explores the Block Floating Point (BFP) hardware scheme with conditional output scaling to handle overflow errors and provide reliable interval arithmetic. *Superpipelining* is applied to the basic architecture to obtain designs with a higher degree of pipelining. The result is a set of designs that can operate at higher clock rates. The choice of the optimal design from this design space is performed by prioritizing throughput based on factors such as clock rate, number of overflows and the number of pipeline stages for the intended application. Throughput is a very important design criterion for signal processing and control applications.

1.1 Description of the Problem

Fixed point implementations in hardware are found to be small in size, involve low implementation costs and have low power consumption as compared to their floating point counterparts when all factors such as the number of bits are kept identical. In spite of these advantages, fixed point designs are plagued by the problem of small dynamic range. A limited dynamic range leads to overflow errors when we represent binary numbers whose bit length exceeds the limits imposed by the chosen number format. In the normal overflow scheme for fixed point two's complement arithmetic, overflow results in *wrap-around*, where attempts to represent a positive number just outside the representable range results in its interpretation as a large negative number, and vice versa [10]. The consequence is that the computed result no longer represents the true value and this makes wrap around a highly undesirable phenomenon.

Figure 1.1 depicts the case for overflow in two's complement fixed point integer arithmetic with a maximum value of *+(N-1)* and a minimum value of *-N*, where N is the limit for number representation with a finite number of bits.

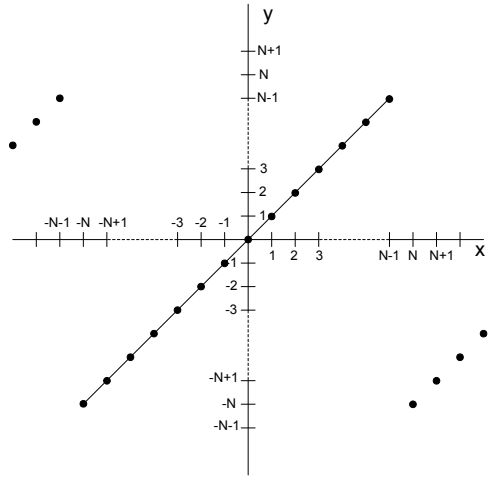


Figure 1.1: Wrap around in two's complement arithmetic.

The definition of an interval is violated when wrap around occurs in fixed point

interval arithmetic. By definition, a closed finite real interval is defined as an ordered set of all real numbers $\{x \in \Re : a \leq x \leq b\}$ lying between and including the real endpoints a and b. *Closed* in this particular context means the endpoints are included as a part of the interval. Therefore, an interval represents a range of real numbers bounded and denoted by its endpoints. Interval operations are performed on the interval endpoints which are represented using fixed point or floating point in computer arithmetic. We present an example that clearly illustrates the outcome of wrap around in the Q7.8 fixed point format, devoting 8 bits to integer representation and 8 bits to the fractional part. By doing this, we aim to highlight the overflow errors that beset the work of [9].

We choose the Q7.8 format in order to match the input data format of the interval ALU of [9]. The interval ALU stored its intermediate interval endpoints in the 32-bit Q15.16 fixed point format and a 16-bit integer output was obtained by performing outward rounding on the fractional part of both interval endpoints. In this experiment, we compute the sum of an interval divergent geometric series $\sum_{n=0}^{N} a^n$ with $a =$ [1.10, 1.15] and N arbitrarily equal to 75 as if it were computed in the interval ALU. Such a computation enables us to observe the fast growth in the magnitude of the terms and wrap around can be observed better.

It is observed that when the magnitude of the upper bound exceeds +32767, it wraps around the positive maximum and assumes a value which is less than the lower bound. Therefore, the result is incorrect and a state of error has been entered since the output interval does not enclose the true result. Figure 1.2 illustrates the incorrect output intervals obtained after overflow.

The above mentioned experiment clearly indicates that an appropriate scheme is needed so that the integrity of interval computations is maintained. We feel the need for firm guidelines to formulate such a scheme and ensure that fixed point interval arithmetic never falls into a state of error. Furthermore, we also feel the need to

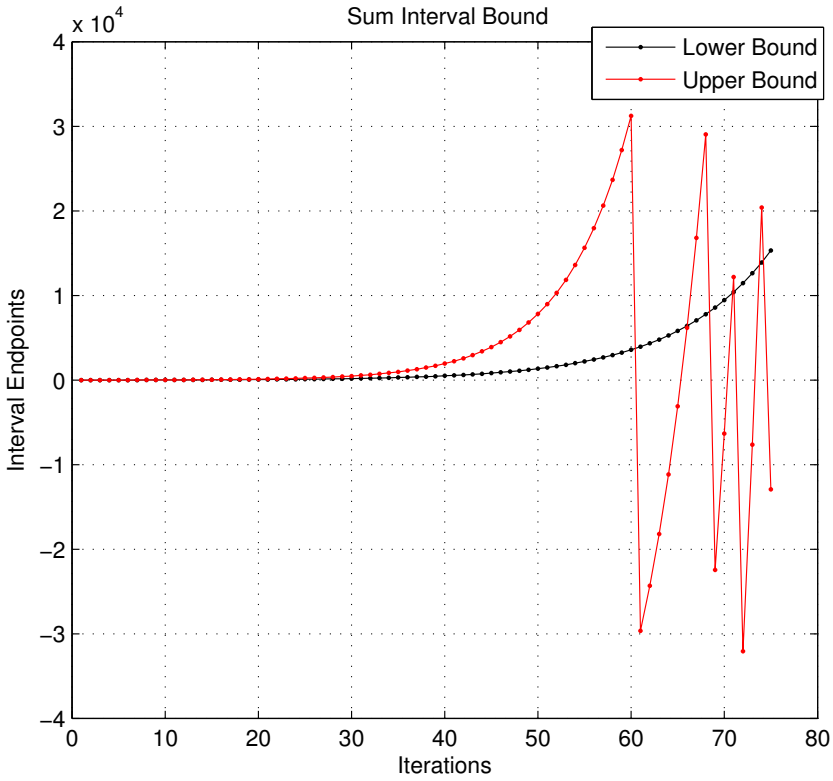


Figure 1.2: Wrap around in divergent interval series (Q7.8 format)

expand the capabilities of the interval ALU [9] so that exact or point-wise evaluations may also be performed in the interval ALU and both theoretical and interval analysis can be performed in the same ALU efficiently. All these factors have led to the development of an interval ALU with Block Floating Point support.

1.2 Background

With the increasing importance of fast execution for interval-based algorithms, the focus has shifted to the development of competitive interval hardware architectures that match the performance of non-interval architectures. While floating point implementations of interval hardware have been carried out previously [8] [11] [12] [13]

as a solution to poor execution rates, there is only one dedicated fixed point interval ALU for DSP and Control applications that has been designed and tested [9]. The fixed point interval ALU has dedicated modules for computing the upper and lower bounds of the interval output and is followed by a dedicated rounding module that performs *outward rounding* on the interval result. 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 [9]. A shortcoming of the design is that overflow errors are not taken into consideration. Therefore, the accumulator overflows over a large time of accumulation leading to unreliable results. The proposed interval ALU, developed in this work, utilizes the skeleton of its architecture from the design of [9].

The proposed interval hardware will adhere strictly to certain *criteria* for reliable results. The work of Van Emden [14] established these criteria as *correctness, totality, closedness, optimality* and *efficiency* for floating point interval arithmetic. These same criteria have been adopted for the *fixed point* implementation since the Block Floating Point (BFP) operations using the proposed interval ALU are performed on the underlying fixed point hardware. These criteria serve as a guideline for making critical decisions pertaining to the hardware design such as the selection of appropriate rounding modes and overflow handling techniques.

Our work explores the architecture of a BFP arithmetic-based interval ALU in order to provide reliable interval operations. BFP arithmetic provides a dynamic range higher than that provided by conventional fixed point representations. In a typical BFP implementation, the input data is divided into non-overlapping blocks along with an integer block exponent term associated with each block. It is possible to detect the magnitudes of data samples in a block of data and then normalize them to bring them all to a common exponent so that fixed point computations may be performed. BFP arithmetic support for point-wise evaluations in most DSPs is typically provided in the form of *Exponent Detection* and *Normalization* instructions [15]. Exponent Detection, when applied to a data sample, provides the number of

redundant sign bits present, thereby indicating its magnitude [16]. A small number of redundant sign bits indicates a large magnitude for the sample, and vice versa. Exponent Detection is performed over a block of data to identify the largest magnitude among all samples and then all the data samples in that block are shifted left by this number. Hence, all data samples in that block are normalized and carry the same exponent. Therefore, fixed point computations can be performed. Normalization implies Exponent Detection followed by left shifting in a single operation [15]. BFP arithmetic has been successfully applied to digital filters [17] and the Fast Fourier Transform [15] [18].

Most commercial DSPs today support BFP operations for point-wise evaluations. Fixed point DSPs such as Analog Devices ASDP-21xx, Texas Instruments TMS320C54x [15], 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 [15]. However, these DSPs do not provide specialized interval arithmetic support with BFP, which is necessary while considering the issues of reliability.

The information on BFP arithmetic is not complete without a mention of the techniques that are used handle overflow errors. The choice of a good overflow handling scheme that also meets the criteria mentioned above is essential to perform reliable fixed point interval arithmetic. The overflow handling techniques are named *Input Scaling* and *Conditional Block Floating-point Scaling* (CBFS) depending upon whether data is scaled *a priori* or *a posteriori* respectively to mitigate the effect of overflows [19] [20] [21]. Each technique is discussed in Chapter 3 in detail.

1.3 Contribution

This thesis makes the following contributions to the current work on fixed point interval hardware architectures. To the best of our knowledge, no other current research has applied the concept of BFP arithmetic to intervals in order to achieve reliable arithmetic.

1. We present an interval ALU with BFP arithmetic support and CBFS for handling overflow errors in fixed point operations

2. We modify the interval architecture to facilitate both point-wise operations for theoretical analysis and interval operations for interval analysis

3. We expand the command set for the interval ALU by introducing logical and comparison operations

1.4 Thesis Organization

This thesis is organized in the following manner: Chapter 2 introduces interval arithmetic and describes its operations. It also describes the criteria that serve as a guideline to implement reliable arithmetic in fixed point hardware. Chapter 3 discusses Block Floating Point arithmetic. It describes two overflow handling techniques associated with BFP arithmetic, namely Input Scaling and Conditional Block Floating-point Scaling (CBFS). Chapter 4 presents the Q0.15 fixed point format and illustrates how to perform fixed point arithmetic, logical, comparison and Block Floating Point operations in this format. Chapter 5 describes the hardware architecture for the interval ALU with detailed module descriptions. It also describes superpipelining as a means of achieving higher throughput. Chapter 6 presents the results of simulation, synthesis and power analysis for the hardware architecture of the BFP interval ALU. Chapter 7 discusses the evaluation of throughput for the BFP interval ALU as a function of the number of overflows, number of pipelined stages and the fastest clock that can be applied to it. It describes the structural hazard observed while

performing point-wise computations in the architecture. This chapter also describes an experiment to perform error analysis on Input Scaling and CBFS to identify the more accurate scheme. In the final chapter, conclusions are drawn from the results obtained and future research is discussed.

Chapter 2

Reliable Interval Arithmetic

Real numbers are of infinite precision while digital machines can only provide limited accuracy on them. By definition, a computer represented set of real numbers $\mathbf{M}$ is a quantized encoding of the elements of a set of real numbers $\Re$. The aim of an optimal computer representation is to maximize the number of elements mapped from $\Re$ on to $\mathbf{M}$ [22] given a restricted number of bits for data representation. Floating point and fixed point representations form two widely used discrete approximations to $\Re$.

Interval arithmetic acknowledges limited precision in computer representation [14] and provides bounds on the error accrued from computations involving discrete approximations. For this reason, it is important that computer arithmetic involving intervals should stay reliable at all times and should not fall into a state of error due to the limitations of number representation in the computer. This chapter discusses the basic interval and set operations followed by a description of a set of criteria that serve as a guideline in designing hardware to perform these operations reliably.

2.1 Interval and Set Operations

The basic interval and set operations include addition, subtraction, multiplication, division, union, intersection, width and midpoint. These operations are important for many applications in signal processing, one example of which is global minimization of cost functions for adaptive IIR filtering [23]. These operations are described below for intervals $I = [r,s]$ and $J = [u,v]$.

A. Interval Addition

$$I + J = [\ r + u,\ s + v\ ]$$

Interval addition involves adding up the corresponding end-points for the input interval arguments. Therefore, the lower end-point for the output interval is the sum of the lower end-points of the input intervals while the upper end-point for the output interval is the sum of the upper endpoints of the input intervals. For example, [1,2] + [4,6] = [5,8]

B. Interval Subtraction

$$I - J = [\ r - v,\ s - u\ ]$$

Interval subtraction involves subtracting the upper endpoint of the second interval from the lower endpoint of the first interval to obtain the lower endpoint for the output interval. Similarly, a subtraction of the lower endpoint of the second interval from the upper endpoint of the first interval yields the upper endpoint for the output interval. For example, [3,4] - [1,2] = [1,3]

C. Interval Multiplication

$$I * J = [\ \min(ru,\ rv,\ su,\ sv),\ \max(ru,\ rv,\ su,\ sv)\ ]$$

This operation can be reduced to a set of conditional operations based upon the signs of the endpoints of the input interval arguments. There are nine possible cases and each of them is formed by a selection of the endpoints being multiplied to yield the output interval endpoints. Table 2.1 shows the various cases associated with this operation.

Table 2.1: 9 cases of interval multiplication

Case	*Description*	*Output Interval Bounds*
1	$x_L \geq 0;\ y_L \geq 0;$	$[x_L y_L, x_U y_U]$
2	$x_L \geq 0;\ y_L < 0 \leq y_U;$	$[x_U y_L, x_U y_U]$
3	$x_L \geq 0;\ y_U < 0;$	$[x_U y_L, x_L y_U]$
4	$x_L < 0 \leq x_U;\ y_L \geq 0;$	$[x_L y_U, x_U y_U]$
5	$x_L < 0 \leq x_U;\ y_U < 0;$	$[x_U y_L, x_L y_L]$
6	$x_U < 0;\ y_L \geq 0;$	$[x_L y_U, x_U y_L]$
7	$x_U < 0;\ y_L < 0 \leq y_U;$	$[x_L y_U, x_L y_L]$
8	$x_U < 0;\ y_U < 0;$	$[x_U y_U, x_L y_L]$
9	$x_L < 0 \leq x_U;\ y_L < 0 \leq y_U;$	$[\min(x_U y_L, x_L y_U), \max(x_L y_L, x_U y_U)]$

An example is [1,4] * [2,3] = [2,12] which is a case where none of the input intervals enclose a zero.

D. Interval Division

Interval Division involves eight cases depending upon whether a zero is contained in the denominator interval or not. The general algorithm used to perform this operation is presented in Table 2.2 and Table 2.3 [7]. It can be implemented similar to multiplication using a set of flags to indicate the choice of operands used in the division operation. However, most DSPs do not provide the divide instruction because this operation occurs very infrequently in signal processing applications. Alternatively, dividing by powers of 2 reduces the operation of division to right shifting operations in fixed point implementations.

Table 2.2: Interval Division A/B with A,B $\in \Re$, 0 $\in$ B

Case	A = [x_L,x_U]	B = [y_L,y_U]	A/B
1	0 $\in$ A	0 $\in$ A	(-∞,∞)
2	0 $\notin$ A	B = [0,0]	[]
3	$x_U < 0$	$y_L < y_U = 0$	[-x_U/y_L,+∞)
4	$x_U < 0$	$y_L < 0 < y_U$	(-∞, x_U /y_U] $\cup$ [x_U /y_L, +∞)
5	$x_U < 0$	$0 = y_L < y_U$	(-∞, x_U/y_U)
6	$x_L > 0$	$y_L < y_U = 0$	(-∞, x_L/y_L,)
7	$x_L > 0$	$y_L < 0 < y_U$	(-∞, x_L /y_L] $\cup$ [x_L /y_U, +∞)
8	$x_L > 0$	[$0 = y_L < y_U$	[x_L /y_U, +∞)

Table 2.3: Interval Division A/B with A,B $\in \Re$, 0 not in B

Case	A = [x_L,x_U]	B = [y_L,y_U]
$x_U < 0$	[x_L /y_L, x_U /y_U]	[x_U /y_L, x_L /y_U]
$x_L < 0 < x_U$	[x_L /y_L, x_U /y_L]	[x_U /y_U, x_L /y_U]
$x_L > 0$	[x_L /y_U, x_U /y_L]	[x_U /y_U, x_L /y_L]

E. Interval Union

Given that I and J are not disjoint, interval union is denoted by

$$I \cup J = [\min(r,\ u),\ \max(s,\ v)]$$

For example, given $I = [2,\ 4]$ and $J = [3,\ 6]$ yields $I \cup J = [2,\ 6]$. This operation is expensive for the case when the sets are disjoint. Throughput is affected if the hardware is expected to put out each disjoint set individually. The ALU developed in this work sets a *disjoint* flag at the end of the operation to indicate disjoint inputs.

F. Interval Intersection

Given that I and J are not disjoint, interval intersection is denoted by

$$I \cap J = [\max(r,\ u),\ \min(s,\ v)]$$

A null set results if the input intervals are disjoint and do not contain any real number elements in common. In this case, the ALU sets a *disjoint* flag at the end of the operation to indicate disjoint inputs. For example, [1,5] ∩ [4,8] = [4,5]

G. Interval Width

$$W(I) = s\text{-}r$$

This operation is performed on a single interval. In the case of the interval I, the width W is given by the difference of the upper bound and the lower bounds.

H. Interval Midpoint

This operation is also performed on a single interval. The midpoint M for the interval I is given by

$$M(I) = (r + s)/2$$

I. Additional Operations

The command set for the interval ALU developed in this work also includes logical, comparison and Block Floating Point (BFP) operations. Logical operations such as bitwise-AND, bitwise-OR and bitwise-XOR have been added from the perspective of a computation engine that performs point-wise operations. Logical operations in DSPs are used widely in applications such as error control coding [24]. The logical operations in the interval ALU operate on the upper and lower bounds for an interval argument. Thus, for interval I, these operations are performed as (s ∨ r), (s ∧ r) and (s ⊕ r) for OR, AND and XOR operations respectively. Comparison operations of two intervals are used to evaluate their minimum and maximum values simultaneously. Applications such as fuzzy adaptive filters based on interval Type-2 systems require such computations [2]. Therefore, for intervals I and J mentioned above, the operations is performed as follows:

$$\min(I,J) = [\min(r,u),\ \min(s,v)]$$
$$\max(I,J) = [\max(r,u),\ \max(s,v)]$$

The command set also contains Block Floating Point operations. These are, however, described later in Section 3.1.2 and 4.4.2.

It has been demonstrated in Section 1.1 that overflow can lead to unreliable interval arithmetic. Overflow occurs when the result of an operation requires more bits on the MSB side for true representation than what is available in the machine [25]. Overflow results primarily from the operations of addition and subtraction. Overflow can also occur with the accumulation of products after multiplication. The cause of overflow is traced to the operation of *addition* in this case. Having introduced the operations to be performed in the interval ALU, we next discuss the criteria, which when met, leads to reliable interval arithmetic.

2.2 Criteria for Reliable Interval Arithmetic

Van Emden [14] has proposed correctness, closedness, totality, optimality and efficiency for the criteria in evaluating interval hardware. We consider a fixed point Interval arithmetic system, based on setting the interval endpoints to finite values, that abides by this set of criteria summarized below:

A. Correctness

An interval operation is said to be *correct* when it yields an output interval containing all the results of point-wise evaluations based on point values which are elements of the argument intervals. For example, if X = [1,2] and Y = [3,5], then this criterion applied to the *addition* operation (+) implies that the resultant interval [4,7] must contain the results of all point-wise additions (x+y) with x $\in$ X and y $\in$ Y.

B. Totality

A *total* interval operation is one that is defined for all possible input arguments. For example, designers face a problem with defining the operation of division (/) when the denominator contains 0. The most common work-around for this is to redefine

the operation of division by excluding 0 from the denominator.

C. Closedness

A *closed* interval operation on the set of real numbers $\Re$ is one that operates on intervals whose endpoints are in $\Re$ and yields an output interval whose endpoints are also in $\Re$. For example, if the operation is multiplication and the input arguments are [1,2] and [3,4], then the result of this operation is [3,8]. Since this operation results in an output interval whose endpoints are real values, we can say that interval multiplication is closed on the set of reals.

D. Optimality

This criterion ensures that the operation is not performing any overestimation and that the bounds are the most optimized ones for the type of representation chosen. The arithmetic should be such that the resultant interval is not wider than necessary. This is applicable to operations such as addition, subtraction, multiplication and division.

E. Efficiency

Efficiency is defined with respect to the implementation of the interval arithmetic in hardware. One way to measure efficiency is through the execution speed which can be improved by eliminating subroutine calls in software or by providing special purpose hardware that deals with the same operation in a much faster way. Efficiency can also be measured in terms of power dissipation or throughput.

It is absolutely essential that arithmetic in the chosen computer representation adheres to the criteria mentioned above for reliable interval computations. The hardware architecture to be presented in Chapter 5 will adhere to these criteria.

Chapter 3

Block Floating Point Arithmetic

Block Floating Point (BFP) is a scaled number representation format similar to floating-point, but its arithmetic operations are performed in fixed point. BFP arithmetic provides a useful tradeoff between the large dynamic-range with the increased hardware complexities of floating point implementations and the limited dynamic range with the relative simplicity of fixed point implementations [26]. Applications for signal processing such as digital filters, calculation of the Fast Fourier Transform and Fast Hartley Transform utilize BFP arithmetic [19].

3.1 Block Floating Point Representation

BFP representation can be considered to be a special case of floating point representation where a block of N numbers has a joint scaling factor corresponding to the maximum magnitude of the numbers in the block. If x_i represents the i^{th} data sample and γ represents the block exponent, then the BFP representation is denoted as

$$[x_1,\ x_2,\ ...,\ \mathrm{x}_N] = [\hat{x}_1,\ \hat{x}_2,\ ...,\ \hat{x}_N\ ]\cdot 2^{\gamma} \text{ where } \hat{x}_i = x_i \cdot 2^{-\gamma}$$

The block exponent γ is defined by

$$\gamma = \lfloor\ \log_2 \mathrm{M}\ \rfloor + 1 + \mathrm{S}$$

where M = max($|x_1|$, ... ,$|x_N|$), $\lfloor . \rfloor$ is the floor function and $|\hat{x}_i| \in [0,1]$. The integer S signifies a constant scaling term for the block exponent which is needed in certain applications like filtering [19].

3.1.1 Data Representation

Data stored in the memory is grouped into non-overlapping blocks of 'N' consecutive samples to perform BFP arithmetic. Each block of data is separately quantized for BFP representation and processed. Figure 3.1 shows the division of an arbitrary data sequence into blocks of size 700.

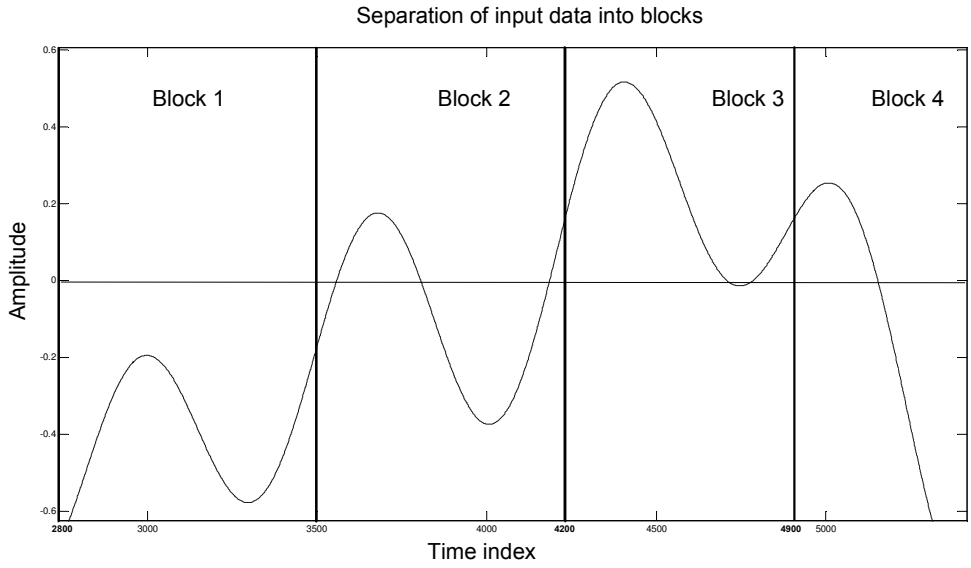


Figure 3.1: Dividing data into blocks

Data processing in BFP arithmetic is performed on a block basis. Within a particular block, all data samples are associated with one common exponent term. From the definition of BFP representation, it is evident that the block exponent γ can be computed from the values of M and S. The use of S is optional and based upon the application. The operation of determining the value of γ for a given data block

is known as Exponent Detection, and the operation of scaling all the data samples in the data block by $2^{-\gamma}$ is known as Normalization. Samples in a data block may be distributed over a wide range of values. Normalizing a data block brings all samples in the data block to a common exponent value and enables fixed point operations to be performed on the samples. It is evident that Exponent Detection and Left shifting by γ boosts the strength of the input data.

3.1.2 Normalization of a data block

Table 3.1 illustrates how to identify the value of γ for a block data size of 4. The data samples are represented in the *decimal* number system and S is fixed at 0. We assume that the dynamic range is [-1,1] and that all data samples are fixed point values which share a pre-normalization common exponent value 'C'. The updated block exponent after normalization will be (C + γ).

Table 3.1: Exponent Detection in a data block of size 4

Data Samples (x)	*Normalized Data* ($\hat{x}$)	$\gamma = (\lfloor \log_{10} \text{M} \rfloor + 1)$
0.189	0.189	M = 0.333 γ = 0
0.214	0.214	
-0.265	-0.265	
0.333	0.333	
0.087	-0.870	M = 0.096 γ = -1
-0.096	0.960	
-0.0014	-0.014	
0.000123	0.00123	

In each data block, the sample with the largest magnitude is chosen to compute the block exponent γ. Therefore, M = 0.333 for the first data block and M = 0.096 for the second block. The value of γ post-normalization is equal to (C) and (C-1) for the first and second data blocks, respectively.

A more intuitive approach to compute the value of γ for samples represented in the decimal system is to assign it with the negated count of the leading zeros for the sample with the largest magnitude in a data block. In the first data block, M = 0.333 does not bear a leading zero digit and hence γ = 0. For the second data block, M = 0.096 bears one leading 0 and hence γ = -1. This scheme can be extended to the binary system where each sample is represented in two's complementary fixed point representation. Here, data samples of small magnitude are fit to the available word size by sign extension. The block exponent γ is assigned the least value of leading redundant sign bits obtained by traversing through all the samples in the data block. Evidently, this value will correspond to the sample with the largest magnitude in the data block. The procedure to normalize the data entails left shifting of all data samples in the entire block of data by γ positions to bring them all to a common exponent value. Henceforth, fixed point operations can be carried out on data samples from this block.

We now present an example that illustrates block normalization for samples represented in two's complement fixed point arithmetic. Consider the example for a block of data comprised of four samples 0.0000100, 0.0011000, 0.0000001 and 0.0001111 in the Q0.7 format. The value of M is identified to be 0.0011000 because it has the least number of leading sign bits. The value of γ is identified to be (-2). Hence, every sample in that data block is left-shifted by two places to bring all samples to a common exponent. Therefore, the data samples bear the values of 0.0010000, 0.1100000, 0.0000100 and 0.0111100 and the number of shifts is (00000010) for an 8-bit exponent. The updated block exponent will be (C-2). The same procedure is applicable to negative numbers as well.

3.1.3 Normalizing an interval data block

Every interval is represented by two endpoints and therefore, two data memory banks must be considered while implementing it in hardware. In this work, we investigate the scenario where normalization is performed such that both endpoints of all intervals in an interval block of data share the same block exponent. Therefore, fixed point operations can be performed on the data directly.

The definition of BFP representation presented in Section 3.1 is extended for 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

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

$$\text{where } [\hat{x}_{Li},\hat{x}_{Ui}] = [x_{Li},x_{Ui}] \cdot 2^{-\gamma}$$

The block exponent γ is defined by

$$\gamma = \lfloor\ \log_2 \text{M}\ \rfloor + 1 + \text{S}$$

where $\text{M} = \max(|x_{L1}|,\ |x_{U1}|,\ ...,|x_{LN}|,\ |x_{UN}|)$ and $|\hat{x}_{Li}| \in [0,1]$; $|\hat{x}_{Ui}| \in [0,1]$. The integer S signifies a constant scaling term for the block exponent and will effect both endpoints in a similar way.

The procedure to identify the block exponent, γ, for a data block comprised of interval data is described next. It is divided into three steps:

1. Identify the minimum number of redundant sign bits among the point-wise data comprised of both endpoints of all the intervals in the block. This helps to identify the largest magnitude-valued endpoint in the interval block.

2. Left-shift both endpoints of every interval in this block by this number.

3. If 'C' was the common block exponent pre-normalization, update the block exponent to a value (C+γ). Store the block exponent in a single location for the interval data block for future reference.

For example, consider two intervals [0.0000110, 0.0010000] and [0.0001101, 0.0001111].

We detect that among these four endpoints, the minimum number of redundant sign bits is present in 0.0010000 and is equal to 2. Therefore, all endpoints are left shifted by 2 positions and the intervals are normalized to the same exponent. Therefore, the normalized intervals are [0.0011000, 0.1000000] and [0.0110100, 0.0111100]. The block exponent for this interval data block is decremented by 2 as compared to the previous exponent.

3.1.4 BFP Hardware Environment

The Block Floating Point Interval Arithmetic and Logic Unit (BFPIALU) developed in this work is intended to be housed in an interval processor with the capability to perform BFP operations. A rough sketch of the environment surrounding the BFPIALU is shown in Figure 3.2.

The data buses for this system are 16-bit bidirectional lines, labeled Data Bus A and Data Bus B, dedicated to the lower and upper endpoints of the interval data respectively. The instructions to be executed are stored in the Code Memory. The Dual Port RAMs labeled A and B comprise the system data memory which are used to store the lower and upper interval data endpoints respectively. They receive data from system I/O data transfers. The Local RAMs, labeled A and B, constitute the local storage for working interval data. These could be useful, for instance, to store the data interval blocks between Exponent Detection and Normalization since the complete procedure requires two traversals through the same data block. The underlying assumption is that Local RAMs are much faster than the Dual Port RAMs. The BFPIALU is the key data processing element in this architecture. It performs arithmetic, logical and BFP operations in fixed point. The Register Files are used for temporary data storage, such as intermediate results from the BFPIALU. They are comprised of a set of high-speed data buffers dedicated to the lower and upper endpoints of the results. The control mechanism is the most complex block in the architecture. Its functions are listed below:

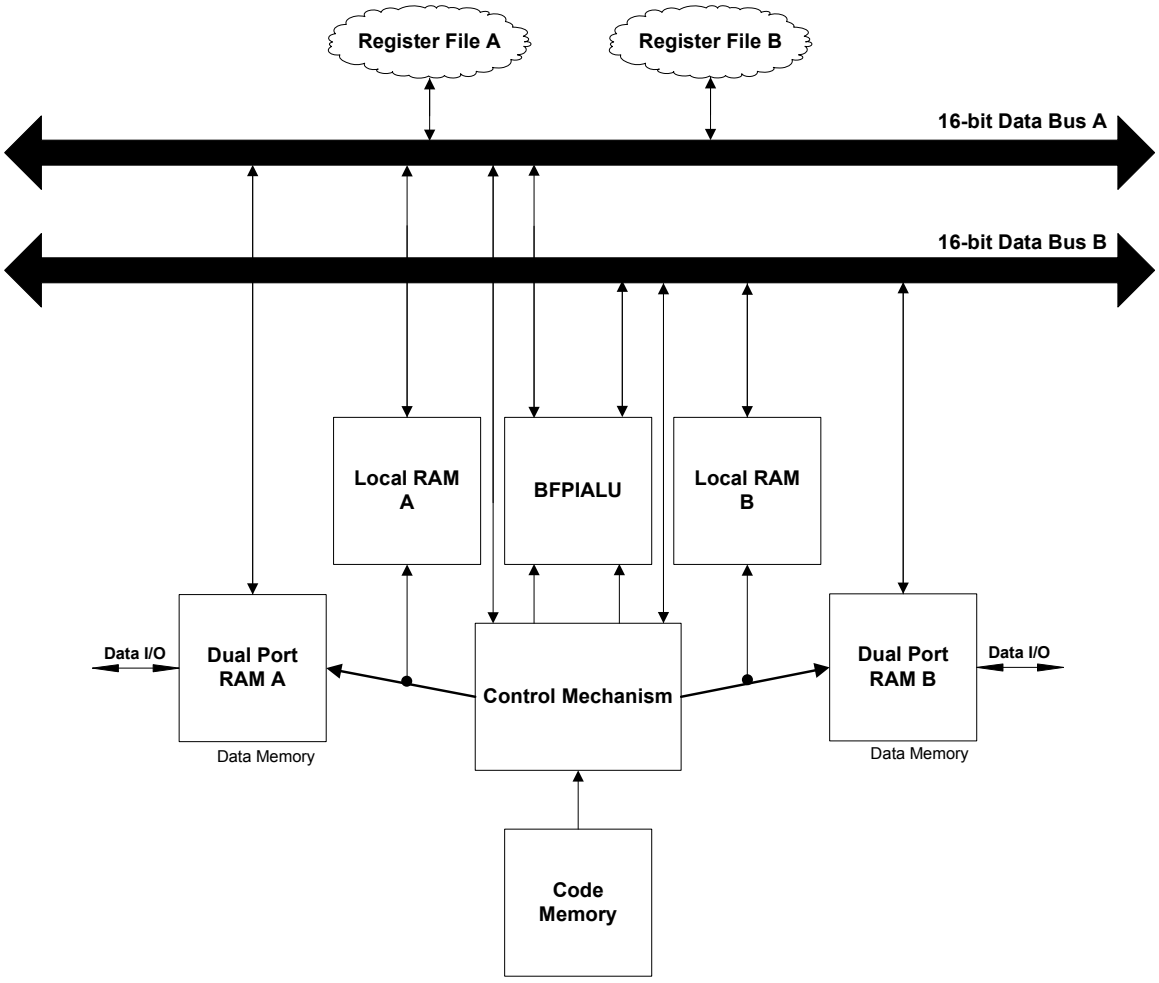


Figure 3.2: Hardware environment for the BFPIALU

- It takes care of fetching instructions from the code memory, decoding them and transferring the data to the appropriate destination.
- It interacts with the Dual Port RAMs and Local RAMs for reading and writing data. It also generates addresses for these operations.
- It feeds the data, applies the necessary command and associated control signals for the BFPIALU.
- It handles the data transfer to and from the Register files.

- Optionally, it houses a DMA mechanism to handle memory interactions involving large amounts of data.

The hardware environment for the BFPIALU may be changed at a later stage suitably to match the requirements of the processor designer. However, care must be taken to ensure that the data feed mechanism and control signals to the BFPIALU remain unchanged.

3.2 Overflow Handling Techniques

Reliability becomes a very significant issue when interval operations are performed in the BFPIALU. The arithmetic performed should be able to cope with and not fall prey to the errors caused by bit growth on the MSB side of the result. Fixed point implementations especially have to deal with overflow errors owing to the availability of a small dynamic range. Since BFP arithmetic is performed in fixed point, additional care must be taken to avoid overflow in BFP arithmetic by adopting appropriate techniques that can either prevent or correct overflow errors.

This work investigates the techniques of pre-operation input data scaling (or *a priori* scaling) and post-operation output data scaling (or *a posteriori* scaling) for a given fixed point operation. The former is known as Input Scaling, whereas the latter is known as Conditional Block Floating-point Scaling (CBFS). A brief description of each of these techniques is presented next.

3.2.1 Input Scaling

The technique of Input Scaling is based on the priciple of *preventing* the occurence of overflow errors. From the definition of BFP provided in Section 3.1, Input Scaling entails a scaling of the input data block using a constant scaling factor 'S' while normalizing it. The value of S is chosen depending upon the number of operations to be performed successively. Unconditional scaling is performed during normalization

so that fixed point additions can freely be performed on the scaled data without any concern of overflow during the actual operation. Input Scaling is fast and simple. All data samples in an output block share the same block exponent, which is equal to the block exponent of the normalized input. This technique is compared to the Conditional Block Floating point Scaling (CBFS) scheme which is discussed next.

3.2.2 Conditional Block Floating-point Scaling

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 erroneous 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 updated to the incremented value of the input block exponent 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.

We next present a listing of the major differences between the techniques of Input Scaling and CBFS:

1. CONSTANT SCALING FACTOR 'S'

   Input Scaling uses a preset constant scaling factor S in anticipation of overflow during normalization. For evaluating dot products, the value of 'S' can be chosen depending upon the number of additions to be performed on the data. In short, the inputs are scaled prior to performing the operation. In contrast,

CBFS implementations do not scale the input data during normalization and set S = 0. Under this scheme, the operation is first performed and the output is scaled down by a factor of two only in the event of overflow.

2. OVERFLOW DETECTION CIRCUITRY

   Since Input Scaling is centered on the idea of *preventing* overflow errors by pre-scaling of inputs, no overflow detection circuitry is required. CBFS implementations are based on the idea of *correcting* overflow errors and hence involve overflow detection circuitry.

3. OUTPUT BLOCK EXPONENT

   The exponent for the output of individual operations such as addition and subtraction for Input scaling is the same as that of the normalized input data. For CBFS implementations that run long iterative operations such as computation of dot products, the output block exponent will change depending upon whether overflow occured or not. Thus, normalizing the output data could consume more time leading to slower implementations.

4. ACCURACY

   Input Scaling implementations record poor accuracy because they scale the input data down unconditionally by a pre-determined factor 'S' before performing the operations. Therefore, such implementations assume the worst-case scenario for every operation - that overflow will occur after every addition. This unconditional scaling lowers accuracy. In contrast, CBFS implementations scale input data by half from the point that overflow actually occurs. The output is scaled only if overflow occurs. This need-based scaling approach leads to more accurate results.

5. SPEED AND COMPLEXITY OF IMPLEMENTATION

   Input Scaling technique is easy to implement and fast in performance. Since the output is obtained at the same block exponent as the input, it can always be fed directly as input to the next stage of computations. Iterative computations such as the evaluation of dot products can be performed in a fast and simple

way, namely through direct fixed point addition. On the other hand, CBFS technique is more complex and involves overflow detection at the end of the operation. Normalizing the output data block for the next stage of processing is time-consuming because different output points may have been scaled differently depending upon whether overflow occurred or not. This is attributed to the fact that the output block exponent increments in value for each overflow that occurs in evaluating the result of a computation such as the evaluation of a dot product. Thus, the output is distributed into groups that share a common block exponent at the end of the processing. Therefore, an additional step of normalizing all data within a block to a common exponent has to be performed if the output has to be fed into the next stage of data processing. When overflows occur very frequently, the complexity of BFP arithmetic using CBFS approaches that of conventional floating point arithmetic in which the results of the operations are normalized after the operation. Therefore, CBFS is a computationally expensive technique.

6. THE OPTIMALITY CRITERION

   Input Scaling leads to an over-estimation of the output interval bounds. Scaling the fixed point interval endpoints, performing outward rounding and then shifting back to the original scale results in an interval wider than the original one. Since scaling is performed in Input Scaling irrespective of the occurence of overflow, a wider output interval results even if no overflow were to occur. Therefore, the criterion of *optimality* described in Section 2.2 is not met by Input Scaling. In contrast, CBFS performs scaling and rounding only if overflow is detected. Hence, it meets the criterion of optimality.

CBFS introduces the lesser distortion (noise) caused by finite word length [21] than Input Scaling. This is advantageous for the BFPIALU since it is intended to be used for signal processing and control applications. Furthermore, CBFS does not over-estimate the output interval unnecessarily leading to optimal interval bounds. Therefore, the design of the BFPIALU, proposed in this work, implements the CBFS technique.

Having decided upon the overflow technique, we next analyse a common overflow scenario that can occur in the BFPIALU. The computation chosen for this example is the evaluation of an *iterative accumulation* of terms.

3.2.3 BFP arithmetic with CBFS Example

We consider the evaluation of a series, S = $\sum_{i=1}^{5} X_i$, where X_i denotes the i^{th} interval. In order to study the sum obtained after each accumulation, we assume that the individual terms of the accumulated interval sum obtained are written out to memory. We describe the operations that are performed in order to compute the final sum of the series using BFP arithmetic. An interval data memory comprised of two memory banks is dedicated to storing the lower endpoint and the upper endpoint of the input intervals. The output endpoints are written out to Register Files A and B. Figure 3.3 shows an overview of the scheme to perform this computation. Simultaneous readouts from successive locations of the first data block from local memory A and B yields input intervals X1, X2, X3, X4 and X5 in order for operation. The interval sum terms obtained are labeled S1, S2 etc.

We assume that the input interval data X1-X5 in Block 1 is normalized to a block exponent 'C' using the procedure outlined in Section 3.1.3. The sequence of operations performed to compute the sum of the series is presented below.

S1 = x1 + x2 : NO OVERFLOW

S1 is sent to the output memory bank. The output block exponent is C

S2 = S1 + x3 : NO OVERFLOW

S2 is sent to the output memory bank. The output block exponent is C

S3 = S2 + x4 : OVERFLOW!!

S3, which is half the overflowed result, is sent to the output memory bank. The

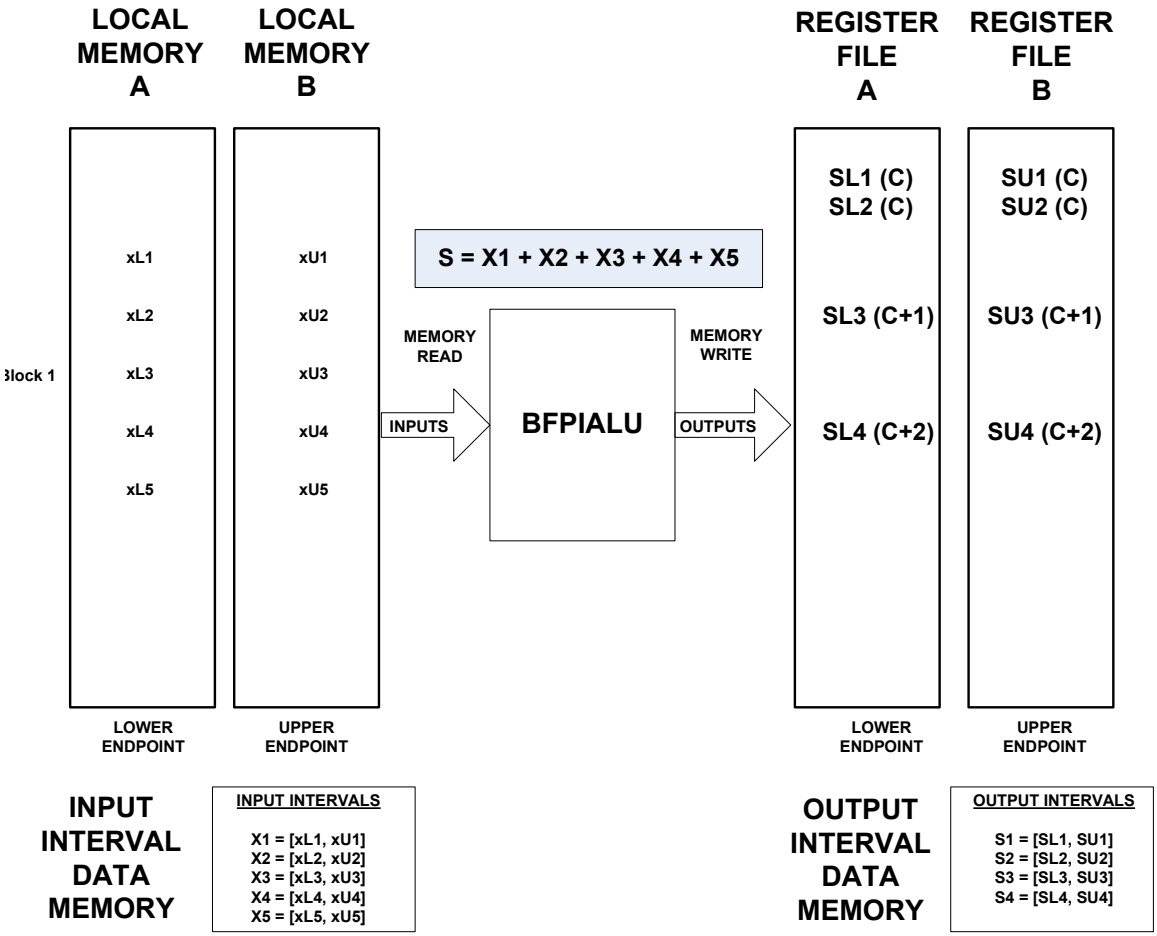


Figure 3.3: Interval BFP system with CBFS

output block exponent is (C+1).

The next addition involves sum S3 which is scaled by a factor of 2 and x5 which has not been scaled. Hence, x5 is scaled down by a factor of 2 and added to S3.

$$S4 = S3 + (x5/2) : \text{OVERFLOW!!}$$

The output block exponent is (C+2). The output is thus segmented into groups of data that bear a common exponent and this scheme devotes one location to store the block exponent per group as seen in Figure 3.3. This example clearly illustrates that a new group of output sum intervals are obtained with each overflow that share a common exponent of (c+γ+p) where p is the number of overflows.

Table 3.2: Grouped Output Points Sharing a Common Exponent

OutputSum	*Block Scale Factor*
S1	C
S2	C
1	
S3	C+1
2	
S4	C+2

It must be noted that whenever overflow occurs, the output block exponent is incremented in value. Thus, the output points looks as shown in Table 3.2. Evidently, the proposed hardware architecture should incorporate input scaling blocks in addition to overflow detection circuitry so that iterative computations may be carried out.

3.2.4 Saturation Arithmetic for Overflow Handling

Systems based on saturation arithmetic saturate the output value to the most positive or the most negative value representable in the event of occurence of overflow, rather than allow a two's complement wrap-around effect to occur. When saturation is applied to interval endpoints that have undergone overflow, the endpoints are clamped to the corresponding maximum limits. Evidently, this leads to an underestimation for the true result of the operation. As a result, the criterion for *correctness* as discussed in section 2.2 is violated. Hence this scheme is not suited for implementing interval arithmetic in hardware.

Chapter 4

Design Specifications

The interval ALU is based on a two's complement fixed point parallel architecture that computes the upper and lower endpoint of the output interval simultaneously. The goal of this chapter is to specify the design specifications and justify design decisions such as the choice of fixed point format for the BFPIALU. This chapter discusses the arithmetic, logical, comparison and BFP operations in the BFPIALU. It also describes the technique of overflow detection for these operations followed by a discussion on the rounding modes.

4.1 Fixed Point Data Format

Fixed-point fractions are denoted using the Q-format where Q denotes *Quantity of Fractional bits* [27]. Qm.n indicates m bits for the integer while n denotes the number of bits devoted to the fractional part. A 16-bit *fraction* is denoted by Q0.15 (or Q.15) while the 16-bit *integer* representation is Q15.0. There is a trade-off between the dynamic range and precision of a fixed point representation. While Q0.15 has the highest precision (2^{-15}) and the least dynamic range (+1), Q15.0 has the highest dynamic range (-32768 to +32767) and the least precision (1.0). Table 4.1 summarizes

the dynamic range and precision of all possible Qm.n formats possible with 16-bit representation [28].

Table 4.1: Dynamic Range/Precision chart for 16-bit Q-formats

Format	*Largest Positive value*	*Least negative value*	*Precision*
Q0.15	0.999969482421875	-1	0.000030517578125
Q1.14	1.99993896484375	-2	0.00006103515625
Q2.13	3.9998779296875	-4	0.0001220703125
Q3.12	7.999755859375	-8	0.000244140625
Q4.11	15.99951191875	-16	0.00048828125
Q5.10	31.9990234375	-32	0.0009765625
Q6.9	63.998046875	-64	0.001953125
Q7.8	127.99609375	-128	0.00390625
Q8.7	255.9921875	-256	0.0078125
Q9.6	511.984375	-512	0.015625
Q10.5	1023.96875	-1024	0.03125
Q11.4	2047.9375	-2048	0.0625
Q12.3	4095.875	-4096	0.125
Q13.2	8191.75	-8192	0.25
Q14.1	16383.5	-16384	0.5
Q15.0	32767	-32768	1

The Q0.15 fixed point format is chosen for data representation in the BFPIALU in order to obtain maximum precision. Figure 4.1 illustrates the Q0.15 two's complement fixed point binary format and the weights associated with each bit position. The binary point is fixed just after the MSB and the remaining bits are devoted to

represent the fraction.

Figure 4.1: Two's complement Q0.15 fixed point representation

4.2 Operations in the Q0.15 format

Operations using the Q0.15 data format are classified into

- Arithmetic operations
- Logical operations
- Comparison operations
- BFP operations

We discuss each category of operations in order starting with the basic arithmetic operations of addition, subtraction and multiplication in the Q0.15 format. We then follow it up with a discussion of Logical, Comparison and BFP operations.

4.2.1 Arithmetic Operations

Arithmetic operations of addition and subtraction face the problem of overflow. The operation of multiplication does not result in bit growth on the MSB side because the product of two fractions is also a fraction. However, overflow errors are faced in the operation of multiply-accumulate. We illustrate these operations first for cases that do not lead to overflow in order to retain focus on the operation performed. We then follow it up with cases that do lead to overflow errors.

A. Arithmetic Operations without Overflow

i) Addition

Addition in Q0.15 involves direct binary addition for the arguments involved.

EXAMPLE 1

1. (0.15625) + (0.78515625) = 0.94140625

0 . 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 (+0.15625) +
0 . 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 (+0.78515625)

---

0 . 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 (0.94140625)

EXAMPLE 2

2. (-0.15625) + (-0.78515625) = -0.94140625

1 . 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 (-0.15625) +
1 . 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 (-0.78515625)

---

ignore
/ **1** . 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 (-0.94140625)

The MSB in Example 2 is ignored because the bit in the MSB position bears the same sign as the inputs.

ii) Subtraction

Subtraction is done by computing the two's complement of the subtrahend and then *adding* it to the minuend in binary. The following examples illustrate this operation.

EXAMPLE 1

1. (0.78515625) - (0.15625) = 0.62890625

0 . 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 (+0.78515625) +

1 . 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 (-0.15625)

---

ignore

~~1~~ **0** . 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 (0.62890625)

EXAMPLE 2

2. (0.15625) - (0.78515625) = -0.62890625

0 . 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 (+0.15625) +

1 . 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 (-0.78515625)

---

ignore

~~0~~ **1** . 0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 (-0.62890625)

We ignore the MSB in the result of both these examples because the input arguments to the addition operation bear different signs and therefore, they do not overflow.

iii) Multiplication

The multiplication of two N-bit two's complement numbers always results in a 2N-bit result. Multiplying two $f_{1,15}$ or Q0.15 numbers produces a product in the $f_{2,30}$ format and this can be reduced to $f_{1,31}$ format by introducing a 0 in the LSB and ignoring the extra sign bit in the actual product. With interval multiplication, outward rounding is required on the interval product to ensure that all possible products for point-wise evaluations with points drawn from the input intervals are contained [9]. Therefore, the excess bits of precision in the lower endpoint of the product are discarded to fit the output word size. The upper endpoint is rounded to $+\infty$ by adding the ORed result of all discarded bits to the remaining bits of the upper end point [9].

The following examples illustrate this operation.

EXAMPLE 1

0. 125 * 0.5625 = 0.0703125

0.001000000000000 * 0.100100000000000

= 00.000100100000000000000000000000

= 0.000100100000000000000000000000**0**

Upon rounding to 16-bits, the product would be 0.000100100000000.

EXAMPLE 2

0.9375 * 0.15625 = 0.146484375

0.111100000000000 * 0.001010000000000

= 00.001001011000000000000000000000

= 0.001001011000000000000000000000

Upon rounding to 4-bits, the product would be 0.010.

The operation of multiplication is performed between positive values only. If any argument is negative, it's two's complement value is taken and then the multiplication is performed. The result is negated again if only one of the input arguments bore a negative value. This step is not undertaken if both arguments are positive or if both arguments are negative. An XOR between the sign bits of the arguments indicates whether the last step is performed or not.

B. Arithmetic Operations with Overflow

Overflow occurs when the true representation of the result requires more bits on the MSB side than the number of bits actually available. The following section presents a method of overflow detection.

i) Overflow Detection Scheme in Addition

Overflow is detected for the *addition* operation involving two arguments x and y when both the following conditions are satisfied: [29]

- Both x and y bear the same sign
- The sign of the output is not equal to the sign of the inputs

The following examples illustrate overflow for addition. Consider two positive arguments 0.75 and 0.5 under the Q0.15 fixed point format. Adding these arguments should lead to an overflow since the sum 1.25 exceeds the highest representable value of $+(1\text{-}2^{-15})$.

EXAMPLE

0 . 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (0.5) +
0 . 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (0.75)

---

1 . 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (-0.75)

The sign of the result has changed with respect to the inputs; the sign of the result is negative while both x and y are positive. This is a clear indicator of overflow and the resulting value is -0.75 which is incorrect. In this case, the additional bit on the MSB side must be captured by using a Guard bit and the whole result should be right shifted by one position. The Guard bit is one additional bit provided to capture single bit growth beyond the MSB of the result. Figure 4.2 shows the scheme to capture the bit growth, This would yield the final result to be 0.101000000000000 which corresponds to 0.625 at half the original scale. An XOR between the signs of the input arguments indicates if they are of the same sign or not. An XOR between the signs of the output and any one of the inputs indicates if the second condition is true or not. The ANDed result of these is used as the overflow flag. The overflow flag is asserted high only if both the conditions are true.

Consider the addition of two negative numbers (-0.5) and (-0.75).

EXAMPLE

1 . 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (-0.5) +

1 . 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (-0.75)

---

1 **0** . 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (+0.75)

Wrap around about the negative extreme (-1) occurs since the result is less than (-1), the smallest number representable. The value of the result is found to be +0.75 which is wrong. Therefore, it is scaled down by a factor of 2 to yield 1.011000000000000 which is (-0.625).

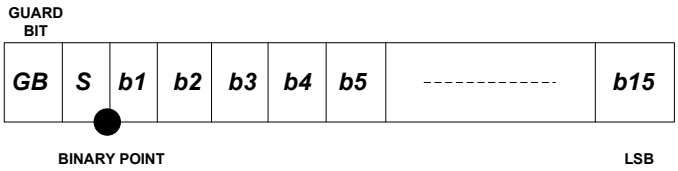


Figure 4.2: Capturing bit growth

The multiply-accumulate instruction performs an addition to accumulate the product terms. Therefore, similar logic for overflow detection is applied to this instruction as well. However, the register size in this case is 33 bits with one guard bit and 32-bit product width.

ii) Overflow Detection Scheme in Subtraction

Given two arguments x and y, overflow is detected in (x-y) if

- Both x and (-y) are of the same sign
- The sign of the result is different from that of x and (-y)

Consider the subtraction (0.75) - (-0.5).

EXAMPLE

0 . 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (+0.75) +

0 . 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (+0.5)

---

1 . 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 (-0.75)

In the example above, overflow is detected since the sign of the result is different from that of the inputs. Therefore, the output needs to be scaled. The actual output scaled by a factor of 2 will be 0.101000000000000 or 0.625 in decimal representation. The value of (-y) is computed by subtracting it from 0. Overflow occurs when both conditions are true, and each condition is verified by an XOR operation between the sign bits if the relevant inputs.

4.2.2 Logical Operations

The logical command set includes the operations of OR, AND and XOR. These operations are widely used for applications such as Error Control Coding. This is demonstrated in the examples below.

EXAMPLE 1

OR operation

1 0 1 0 1 0 1 1 0 1 0 0 1 1 1 1

0 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0

---

1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1

EXAMPLE 2

AND operation

1 0 1 0 1 0 1 1 0 1 0 0 1 1 1 1

0 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0

---

0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0

EXAMPLE 3

XOR operation

1 0 1 0 1 0 1 1 0 1 0 0 1 1 1 1

0 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0

---

1 1 0 0 1 1 0 0 0 1 0 1 0 1 1 1

4.2.3 Comparison Operations

Comparison operations are encountered frequently in applications such as fuzzy adaptive filtering [2]. They involve a comparison of corresponding endpoints of the argument intervals. The method of comparing two Q0.15 numbers to identify the minimum among them is discussed next. The other number is the maximum.

Between a positive number and a negative number, the minimum is identified as the one with the sign bit '1'. Between two positive or two negative numbers, an unsigned comparison of the numbers yields the minimum of the two.

EXAMPLE 1

min(1.010101101001111, 0.010001100001000) = 1.010101101001111

EXAMPLE 2

min(1.111000000000000, 1.010000000000000) = 1.010000000000000

4.2.4 BFP Operations

The ALU provides BFP support in the form of Exponent Detection, Normalization and Signed Left shifting operations. These instructions aid the processes of determining the block exponent and block normalization which have been described in Section 3.1.2. These are illustrated here.

A. EXPONENT DETECTION

This section illustrates the operation of Exponent Detection in which was described in Section 3.1.2. Exponent Detection provides an integer result corresponding to the count of the leading duplicated sign bits in the input. This helps us to identify the magnitude of the largest sample in the data block being operated upon. The data sample that provides the smallest integer as the output of this operation is identified to be the sample with the largest magnitude, see examples provided below.

EXAMPLE

1. EXP_DET(0.000 0000 0001 0110) = 10
2. EXP_DET(1.111 1111 0100 1010) = 7

The duplicated redundant sign bits in each example considered are underlined. It is seen that the magnitude of the data in Example 2 is more than the magnitude of the value in Example 1. Therefore, the exponent detected for Example 2 is less than the exponent detected for Example 1.

B. SIGNED LEFT SHIFTING

The process of normalizing a block is divided into two steps, namely Exponent Detection and Left Shifting. First, Exponent Detection is performed on all samples in the data block as mentioned above. The least integer obtained is noted and then all samples in the block are left-shifted by this number. The operation of signed left shifting retains the sign of the value that is left shifted while shifting every other bit left by the required amount. Consider the following examples.

EXAMPLE

1. LEFT SHIFT BY 2 POSITIONS: 0.000 0100 0101 0110 = 0.001 0001 0101 1000
2. LEFT SHIFT BY 4 POSITIONS: 1.111 1111 1111 0101 = 1.111 1111 0101 0000

The block exponent for a data block is an integer. The same Q0.15 fixed point format ALU developed in this work can also be used for integer operations. However, the limit on the size of the exponent is 7FFF (hex) on the positive side and 8000 (hex) on the negative side. Moreover, scaling of inputs should be inhibited when this operation is performed. Thus, the block exponent can be carried through the BFP operations.

C. DIRECT NORMALIZATION

Normalization refers to the two-step process of Exponent Detection followed by left shifting. Given the scenario where we must normalize a block of data knowing that all samples in it have the same number of leading sign bits, we can avoid Exponent Detection and perform Direct Normalization. This operation directly eliminates all redundant sign bits in a sample when applied to it. This is illustrated in the examples below:

EXAMPLE

1. NORM(0.000 0000 0001 0110) = 0.101 1000 0000 0000
2. NORM(1.111 1111 1100 1010) = 1.001 0100 0000 0000

The underlined bits are eliminated directly once Normalization is applied to these data samples.

4.3 Outward Rounding

In most systems, fixed sized registers impose a constraint on the number of bits that the output of an operation can occupy. The word length of the outputs should be trimmed to fit the size of the registers. A proper choice of rounding scheme is

essential to minimize errors since this implies the introduction of precision rounding errors [9]. With interval arithmetic, it is of utmost importance that the selected rounding scheme results in output intervals that do not under-estimate the true value of the computation. One way to ensure this is to use Outward Rounding where the Lower Bound is rounded towards -∞ and the Upper Bound is rounded to +∞ value.

This work retains the Outward rounding scheme as discussed in [9] because it satisfies the property of *optimality* by not overestimating the interval bounds as stated earlier in Chapter 2. The directed rounding scheme for interval arithmetic involves rounding to -∞ on the lower bound endpoint and rounding to +∞ on the upper bound endpoint [9].

4.3.1 Rounding to -∞

Rounding to -∞ is the method of representing a high precision value by its nearest smaller value of lower precision in machine representation. The next lower number is evaluated by simply *dropping* the excess bits of precision in two's complement fixed point arithmetic. The following examples illustrate rounding to -∞ for fixed point data in the Q0.15 format.

EXAMPLE
0.001001011000000 is the binary equivalent of 0.146484375 in Q0.15 format. In Q0.7 binary format, it is represented as 0.0010010. This is the binary equivalent of 0.140625 and is obtained by discarding the latter 8-bits of precision.

EXAMPLE
1.1101010 is the binary equivalent of -0.171875 in Q0.7 format. In Q0.3 format, this would be 1.110 or -0.25 in decimal representation.

4.3.2 Rounding to $+\infty$

Rounding to $+\infty$ involves the addition of an LSB in the resultant Q-format to account for all the discarded bits which have a finite binary value. This rounding scheme yields a quantity which has lower precision but is the smallest value representable in the new Q-format bearing a value greater than or equal to the former value. It can be applied to both positive as well as negative numbers. Using all discarded bits to decide the addition of the LSB is better than normal rounding where only the MSB of the discarded bits is used. This is because using all discarded bits provides more preciseness in rounding than using merely the MSB.

EXAMPLE
0.0010010 is the binary equivalent of 0.140625 in the Q0.7 format. When rounded to $+\infty$ in Q0.3 format, we get 0.010 which corresponds to 0.25 in the decimal representation.

EXAMPLE
1.1010101 is the binary equivalent of -0.3359375 in Q0.7 format. When rounded to $+\infty$ in Q0.3 format, we get 1.110 which corresponds to -0.25 in the decimal representation.

Chapter 5

Hardware Architecture

This chapter provides a description of all the hardware modules that constitute the BFPIALU. It describes the functions for the individual modules and the interactions between them. The chapter also gives details of the logic design at the gate level for each module. Several pipelined designs based on the proposed architecture are built in order to attain higher throughput by reducing the logic depth of the critical path in the design. The design is built at the Register Transfer Level (RTL) level using Verilog [30]. This design conforms to the architecture of the non-pipelined version. It is then pipelined to varying depth to attain higher throughput.

5.1 Overview of the Architecture

The architecture of the proposed BFPIALU comprises of four main modules, namely the *Flag Generator* module, the *Lower Bound* module, the *Upper Bound* module and the *Scale Synchronizer* module. The Flag Generator generates various control signals which are used by the Lower and Upper Bound modules for executing the commands. As the names suggest, the Lower Bound and the Upper Bound modules perform computations to evaluate the corresponding bounds of the output

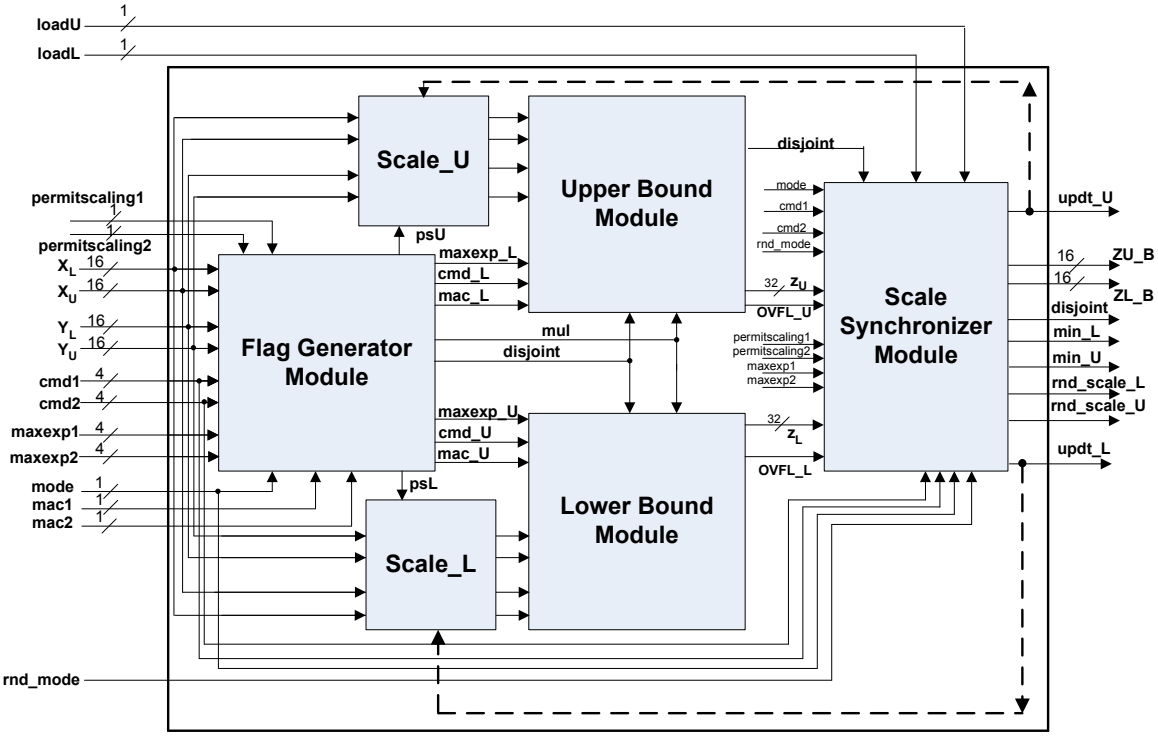


Figure 5.1: Top Level Block Diagram

interval. Both the modules scale their outputs down by a factor of 2 whenever overflow occurs. The *Scale Synchronizer* module performs the synchronization of the scaling since the hardware has to output an interval whose bounds are all at the same scale. It also updates the increment in the output block exponent value upon overflow. The modules *Scale_L* and *Scale_U* are minor logic blocks that perform shifting of the inputs and are included as part of the Lower Bound and Upper Bound modules respectively. These are discussed after the Scale Synchronizer has been described. Figure 5.1 illustrates the interconnection of the modules in this architecture.

Table 5.1 shows the inputs to the interval ALU and their corresponding bit-widths. 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 *point̄wise* 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 *point̄wise* mode of operation, both modules can execute different commands independently. This explains the presence of two copies of command entity *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. Asserting the *mac* 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.

Table 5.2 provides a description of the outputs of the interval ALU. 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. A further description of the output lines is provided while describing the individual modules.

Table 5.3 lists the various operations that can be performed in the interval ALU. While commands 0-7 have been implemented in [9], 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*

Table 5.1: Architecture Input Description

Input	*Description*	*Bits*
X_L	Lower bound of Interval X	16
X_U	Upper bound of Interval X	16
Y_L	Lower bound of Interval Y	16
Y_U	Upper bound of Interval Y	16
mode	Operation Mode : *Interval* / *Pointwise*	1
cmd1	Operation to be performed in the Lower Bound	4
cmd2	Operation to be performed in the Upper Bound	4
maxexp1	Positions to left shift X_L in Lower Bound	4
maxexp2	Positions to left shift Y_L in the Upper Bound	4
mac1	Performs MAC operation in Lower Bound	1
mac2	Performs MAC operation in Upper Bound	1
permitscaling1	Iterative computations in Lower Bound	1
permitscaling2	Iterative computations in Upper Bound	1
loadL	Load block exponent increment for the Lower Bound	1
loadU	Load block exponent increment for the Upper Bound	1
rnd_mode	Rounding mode for *Pointwise* mode	1

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.

Table 5.2: Architecture Output Description

Output	*Description*	*Bits*
ZL_B	Lower bound of Output interval	16
ZU_B	Upper bound of Output interval	16
min_L	Min Exponent Detected over a data block in the Lower Bound	1
min_U	Min Exponent Detected over a data block in the Upper Bound	1
rnd_scale_U	Output Flag to indicate scaling to counteract overflow due to rounding to $+\infty$ in the Upper Bound module	1
rnd_scale_L	Output Flag to indicate scaling to counteract overflow due to rounding to $+\infty$ in the Lower Bound module	1

The following sections describe each module of the interval ALU architecture in detail shown in Figure 5.1.

5.1.1 Flag Generator Module

Figure 5.2 provides an overview of the Flag Generator module. It is comprised of three independent functional blocks, namely *Flag Generation for interval multiplication*, *Disjoint flag generation for interval union and intersection* and the *Control Distribution Unit*. The functions of the Flag Generator module are listed below:

- Identifying the appropriate case of interval multiplication to be performed
- Identifying and flagging disjoint input intervals
- Distributing appropriate control signals to the Lower and Upper Bound modules depending on the mode of operation

The following sections explain the functionality and logic in each of these blocks in detail.

Table 5.3: Command Set for the ALU

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

A. FLAG GENERATION FOR INTERVAL MULTIPLICATION

The actual endpoints to be multiplied for interval multiplication depends upon the signs of the input interval endpoints. Interval multiplication consists of nine cases, as listed in Table 5.6. A set of six flags are used to form a 4-bit signal *mul* which indicates which case of multiplication is to be performed. Table 5.4 indicates the implication for each flag when asserted high. Table 5.5 illustrates the relation between the flags and *mul*.

The selection logic to implement the relation between the flags and *mul* in hard-

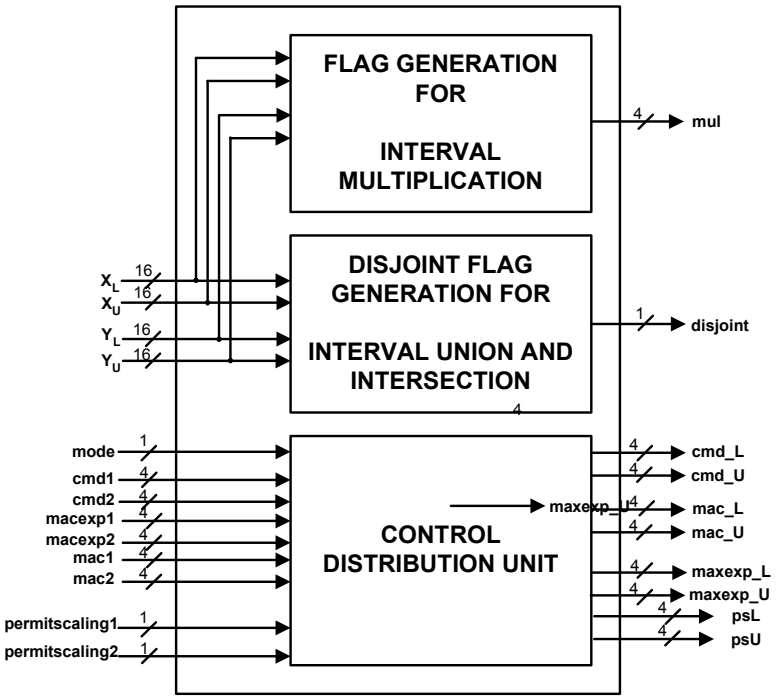


Figure 5.2: Flag Generation Module

Table 5.4: The Scheme of Flags

Flag	*OPERATION*
flag1	$X_L \geq 0$
flag2	$Y_L \geq 0$
flag3	$X_U < 0$
flag4	$Y_U < 0$
flag5	$X_L < 0$ and $X_U \geq 0$
flag6	$Y_L < 0$ and $Y_U \geq 0$

ware is adopted from [9]. Changes are incorporated wherever *pointwise* operations are concerned. Table 5.6 illustrates the one-to-one mapping between various cases of multiplication and the corresponding value of *mul*.

Table 5.5: Deriving *mul* from the Flags

Combination of Flag	*mul*
flag1 & flag2	1
flag1 & flag6	2
flag1 & flag4	3
flag5 & flag2	4
flag5 & flag4	5
flag3 & flag2	6
flag3 & flag6	7
flag3 & flag4	8
NONE	0

Table 5.6: *mul* signal for the different cases of interval multiplication

mul	*Description*	*Output Interval Bounds*
0001	$x_L \geq 0;\ y_L \geq 0;$	$[x_L y_L, x_U y_U]$
0010	$x_L \geq 0;\ y_L < 0 \leq y_U;$	$[x_U y_L, x_U y_U]$
0011	$x_L \geq 0;\ y_U < 0;$	$[x_U y_L, x_L y_U]$
0100	$x_L < 0 \leq x_U;\ y_L \geq 0;$	$[x_L y_U, x_U y_U]$
0101	$x_L < 0 \leq x_U;\ y_U < 0;$	$[x_U y_L, x_L y_L]$
0110	$x_U < 0;\ y_L \geq 0;$	$[x_L y_U, x_U y_L]$
0111	$x_U < 0;\ y_L < 0 \leq y_U;$	$[x_L y_U, x_L y_L]$
1000	$x_U < 0;\ y_U < 0;$	$[x_U y_U, x_L y_L]$
0000	$x_L < 0 \leq x_U;\ y_L < 0 \leq y_U;$	$[\min(x_U y_L, x_L y_U), \max(x_L y_L, x_U y_U)]$

B. FLAG GENERATION FOR DISJOINT INTERVAL OPERATION

The Flag Generation module houses the logic to identify if the input intervals are disjoint or not. Two intervals [x_L, x_U] and [y_L,y_U] with real endpoints are disjoint if either $x_U < y_L$ or $y_U < x_L$. This is indicated in Figure 5.3.

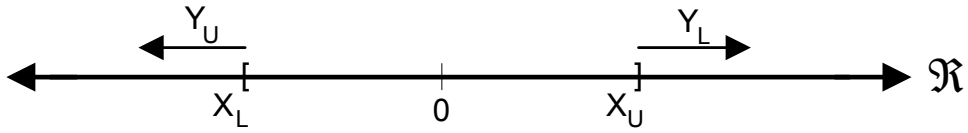


Figure 5.3: Disjoint intervals X and Y

The design makes use of a *comparison* logic block to compare interval endpoints which are in the Q0.15 format. Figure 5.4 depicts the hardware used to perform this comparison. The block indicates whether or not a signed Q0.15 number x is less than a signed Q0.15 number y. If x and y are of different signs and the sign of x is negative, then the output is flagged 1. If not, it is flagged 0. However, if both x and y are of the same sign, then they can be compared directly as unsigned binary numbers. In this case, the output is flagged 1 if *unsigned* x is less than *unsigned* y. If not, it is flagged 0.

The *comparison* logic block is used twice in this module, once to compare x_U with y_L and then to compare y_U with x_L. The results of these comparisons are OR-ed to produce the *disjoint* signal. Both the Upper Bound and the Lower Bound modules utilize this signal during the execution of *union* and *intersection* operations. If *disjoint* is high, then the output of these operations in the interval ALU is 0 while the flag is sent out as an output status line.

C. CONTROL DISTRIBUTION UNIT

The Flag Generation module distributes proper commands and signals to both Upper and Lower Bound modules depending upon the mode of operation of the

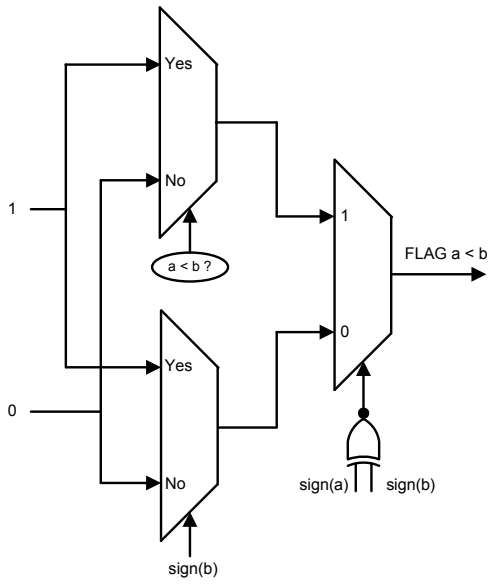


Figure 5.4: Comparison Unit to compare two Q0.15 numbers

ALU. As mentioned earlier, the ALU has two command inputs, *cmd1* and *cmd2*. When the ALU operates in the *interval* mode, both Upper and Lower Bound modules have to execute the same command. Hence, command *cmd1* is supplied to both the modules for an interval operation. When the ALU operates in *pointwise* mode, both the modules should work independently and hence should be able to take individual commands. Thus, command *cmd1* is supplied to the Lower Bound module while command *cmd2* is supplied to the Upper Bound module. The logic to handle the selection of the command input to the Upper Bound module comprises of a simple multiplexor which selects only one signal, *cmd1* or *cmd2*, depending on the status of the *mode* signal. This is illustrated in Figure 5.5.

Similar logic is applied to the selection of *mac1* and *mac2* signals. Since a MAC operation in the *interval* mode implies a simultaneous accumulation of product interval endpoints in both Upper and Lower Bound modules, a common mac signal must be given to both the modules. In this case, signal *mac1* is fed to both the modules. However, in the *pointwise* mode, signal *mac1* is fed to the Lower Bound

module and signal *mac2* is fed to the Upper Bound module. Similar select logic is also applied in the selection of *maxexp2*. Here, the signal *macexp2* is fed to the Upper Bound modules in both modes of ALU operation whereas the Lower Bound module receives *macexp2* if the ALU is in the *interval* mode and *macexp1* if the ALU is in the *pointwise* mode of operation. The status of the *mode* input line forms the basis for performing the selection of these signals for the Lower Bound and Upper Bound modules. The hardware to perform this operation is depicted in Figure 5.5. A similar scheme is also applied to the *permitscaling1* and *permitscaling2*.

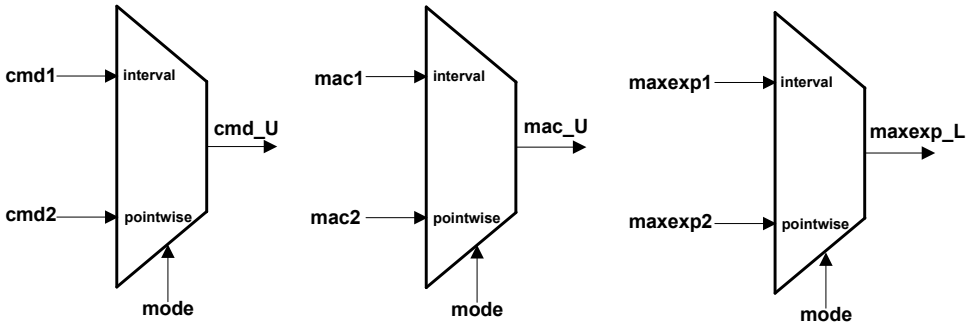


Figure 5.5: Command, MAC and *maxexp* signal Select logic

5.1.2 Lower Bound and Upper Bound modules

The operations in both the Lower and Upper Bound modules are performed in parallel with one logic path dedicated to each operation. The set of operations discussed here pertains to the collective union of the basic interval and set operations described in Chapter 2 and the logical, comparison and BFP operations described in Chapter 4. It is imperative that these operations be provided in this architecture because this makes the architecture a suitable hardware platform to execute interval algorithms *efficiently*. In addition to the new commands added to enhance the utility of this architecture, changes in these modules as compared to the work of [9] include elimination of the *special multiplication block* and the *next* signal for set operations.

All these points are discussed in the ensuing sections. The overall block diagram of the Lower Bound module is shown in Figure 5.6. Additional details pertaining to each command are given below. The inputs to this module are the outputs of 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.

A. Addition, Midpoint, Subtraction, Width

The computation of the midpoint of an interval requires an addition to be performed. So the adder in the *addition* path is reused to perform this operation. A multiplexor is used to feed the right inputs to the adder. A multiplexor selects either a signed right-shifted version of the sum if the command is *midpoint*, or it selects the sum directly if the command is *addition*. The only point to be noted is that the overflow detection circuitry, which is modeled along the method mentioned in Section 4.3.1, halves the result in the case of overflow. Hence, depending upon whether overflow occurs or not, the input to the final multiplexor is either half the value of the sum or the sum itself. The overflow detection circuitry following the adder outputs a signal *overflow_add* to flag overflow. When asserted high, it indicates overflow. The reason for this is that the midpoint operation can never face overflow because the result is always between the endpoints of the interval. Figure 5.7 illustrates the logic design for this path.

A similar optimization is applied to the *subtraction* and *width* commands as well. Figure 5.8 illustrate the logic in the addition/midpoint and subtraction/width command paths. Unlike the operation of *midpoint*, the operation of *width* is susceptible to overflow. The output detection scheme is mentioned in Section 4.3.2.

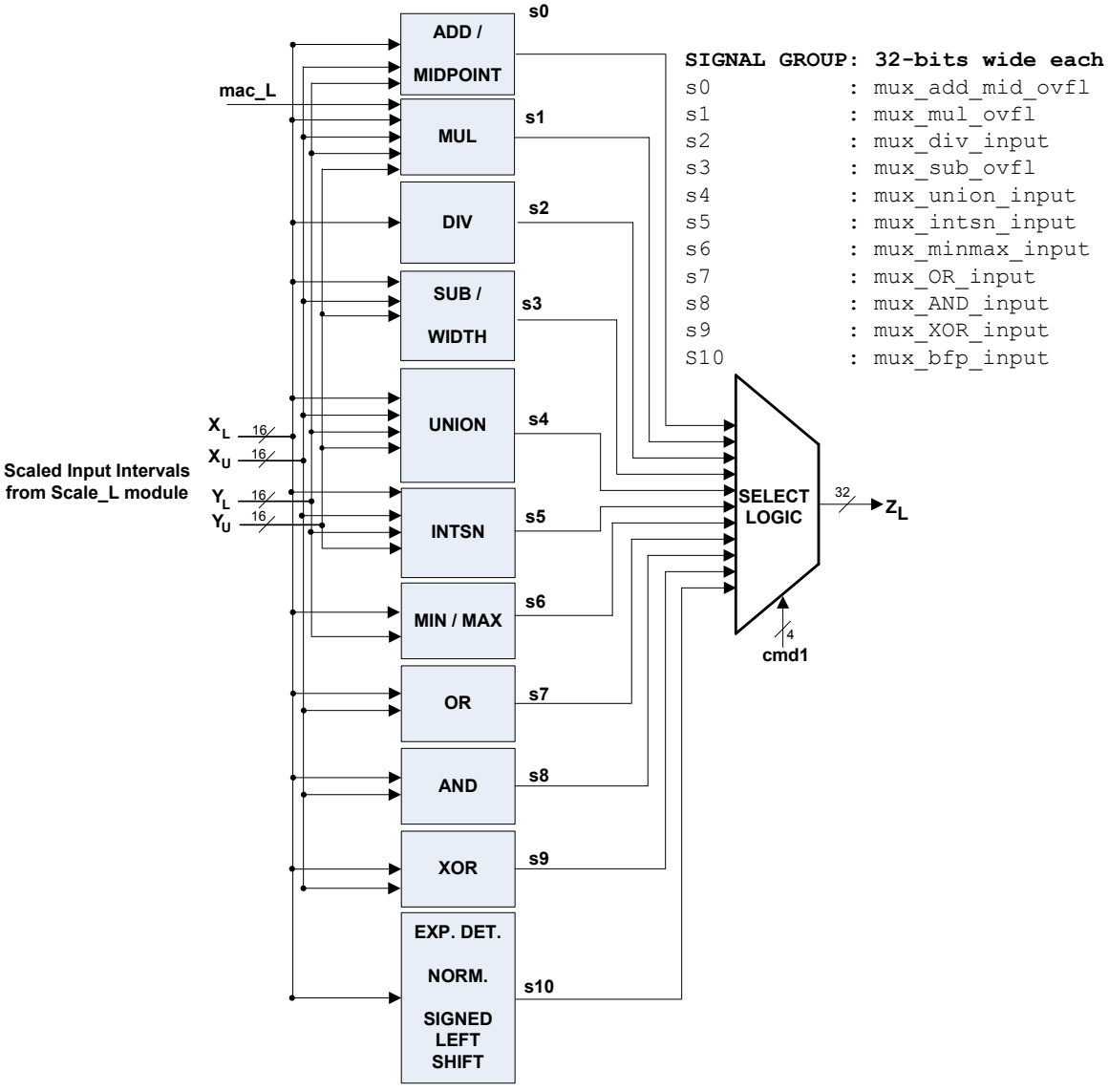


Figure 5.6: Lower Bound Module

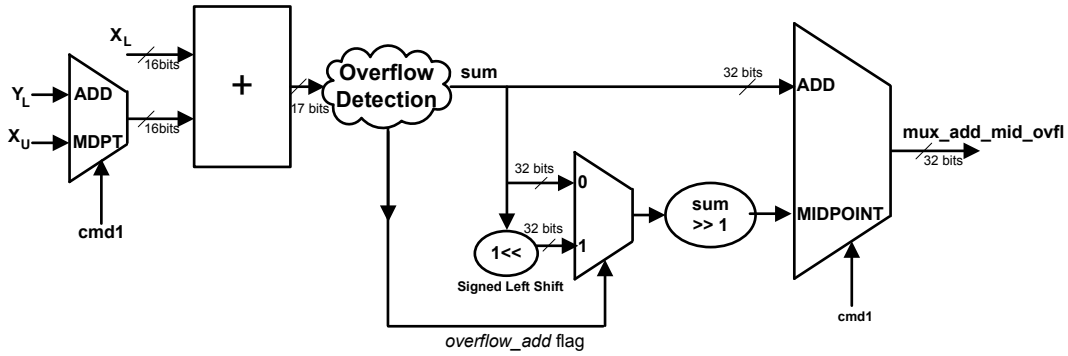


Figure 5.7: Multiplexed Addition and Midpoint operation

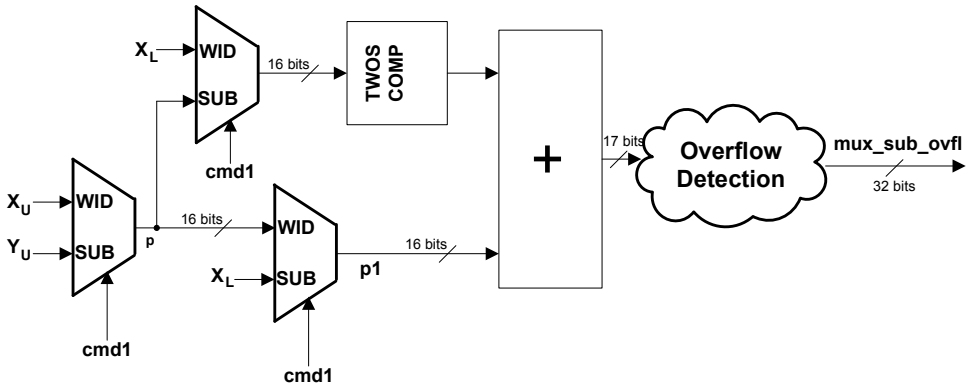


Figure 5.8: Multiplexed Subtraction and Width operation

B. Multiplication

Figure 5.9 illustrates the hardware for the multiplication path. In order to reduce the period of execution for the special case of multiplication to one cycle, an additional multiplier is used. It plays a role only if signal *mul* from the output of the Flag Generator module bears a value 0000. This corresponds to multiplication where both intervals enclose 0 between them. Based on the value of *mul*, the selection logic chooses the appropriate interval endpoints to be multiplied. The two's complement logic converts the negative operands to positive ones so that the multiplication takes place between positive arguments. The result is again passed through the two's complement circuitry and the product is negated if any one of the operands was negative.

In the normal case of multiplication, signal *prod1* is sent to the accumulator. For the special case however, the minimum of *prod1* and *prod2* is identified and sent to the accumulator. Signal *product_L* signifies the product of the multiplication operation.

The Multiply Accumulate Block

Figure 5.10 illustrates the hardware that performs the accumulation operation. In the accumulation mode, the value stored in output register, denoted by z_L in the figure, is added to the current product, namely *product_L*, and the sum thus obtained is stored in z_L again. This operation is known as *accumulation*. It is particularly useful in the evaluation of dot products. If the ALU is not in the accumulation mode (indicated by *mac* signal being low), then the product is passed on to z_L. To perform this operation, *product_L* is added with zeros. A multiplexor selects out either the value of z_L or zeros depending on the status of the *mac* signal for the module. The output of this path is denoted by signal *mux_mul_ovfl*.

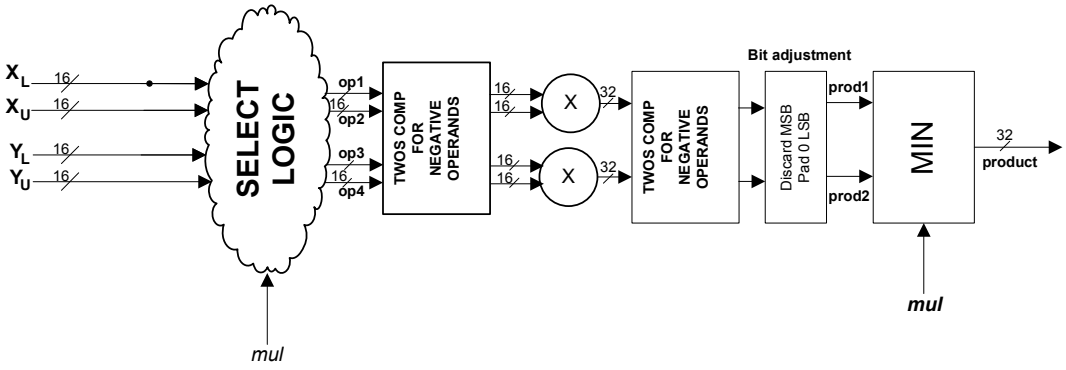


Figure 5.9: Hardware for Multiplication in Lower Bound module

Hardware for Overflow Detection

Addition, *subtraction* and *multiply-accumulate* incorporate overflow detection modules in their data paths. The method used to detect overflow in arithmetic operations

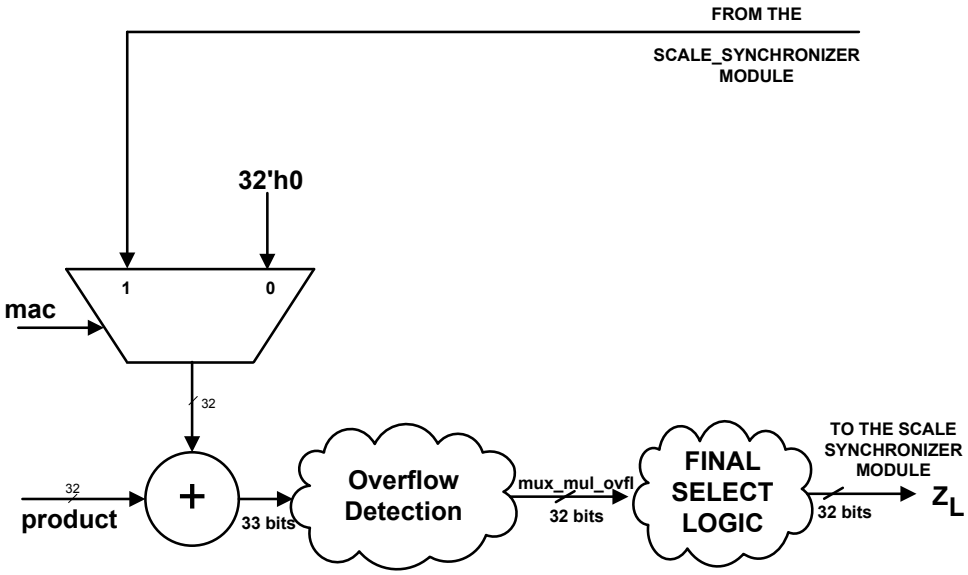


Figure 5.10: Multiply-Accumulation scheme in hardware

is mentioned in section 4.2.1. Overflow detection for multiply-accumulate follows the same rules as used for addition.

An extra zero is padded on the MSB side to capture the bit growth as a result of these operations. The similarity or dissimilarity in input sign bits can be identified using *xor* logic between them. Overflow is detected by following the algorithm mentioned in Section 4.2.1 for addition and subtraction. The multiply-accumulate operation uses the same algorithm as that used by the operation of addition. If overflow is detected in the result of the operation, then it is shifted right to scale it down by a factor of 2. A flag, OVFL_L, is then asserted high to indicate the scaled status of the output to the Scale Synchronizer module. The operations of addition, subtraction and multiply-accumulate have their own dedicated overflow flags. However, only one overflow flag corresponding to the operation under execution is sent to the Scale Synchronizer module using select logic to minimize communication between the modules. If no overflow is detected, then no flag is asserted high and the results are not scaled down. This way, conditional scaling is performed on the output. Figure 5.11 illustrates the block diagram of this module. The *Conditional Shift* block represents

the overflow detection schemes for addition, subtraction and multiply-accumulate. It is a multiplexor that selects out a right shifted version of the output if overflow is detected or simply passes the output through if no overflow occurs. The status of overflow in the operation is denoted by the signal *overflow_detect* and it is forwarded to the Scale Synchronizer module as OVFL_L. The Upper Bound module uses an overflow flag OVFL_U, similar to OVFL_L.

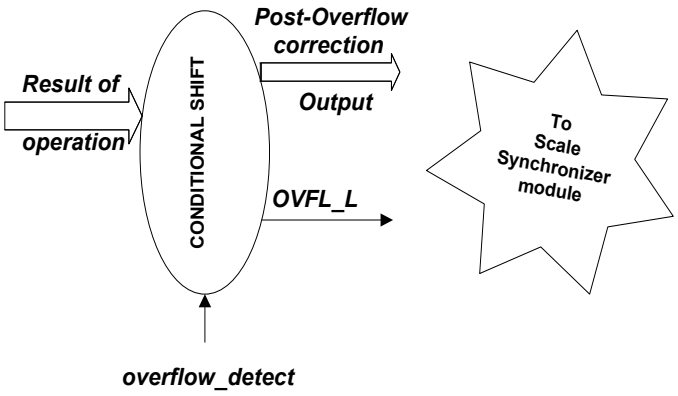


Figure 5.11: Hardware for overflow detection

C. Division, Union, Intersection, Min, Max, Left Shift

DSPs do not usually provide the general *divide* command because it occurs very infrequently in signal processing applications and it can increase area significantly. A limited divide operation is implemented by providing the facility to shift the input data right, which corresponds to dividing the data by 2. This ALU also supports division in powers of 2.

Union and *intersection* operations are performed using the comparison module shown in Figure 5.4. Figure 5.12 illustrates the logic design for this path. *Min* and *Max* operate on X_L and Y_L. As the name of the command suggests, Min is used to find the minimum of the lower bounds of two intervals [X_L,X_U] and [Y_L,Y_U]. Similarly, Max is used to identify the maximum of the lower bounds of the two intervals. The

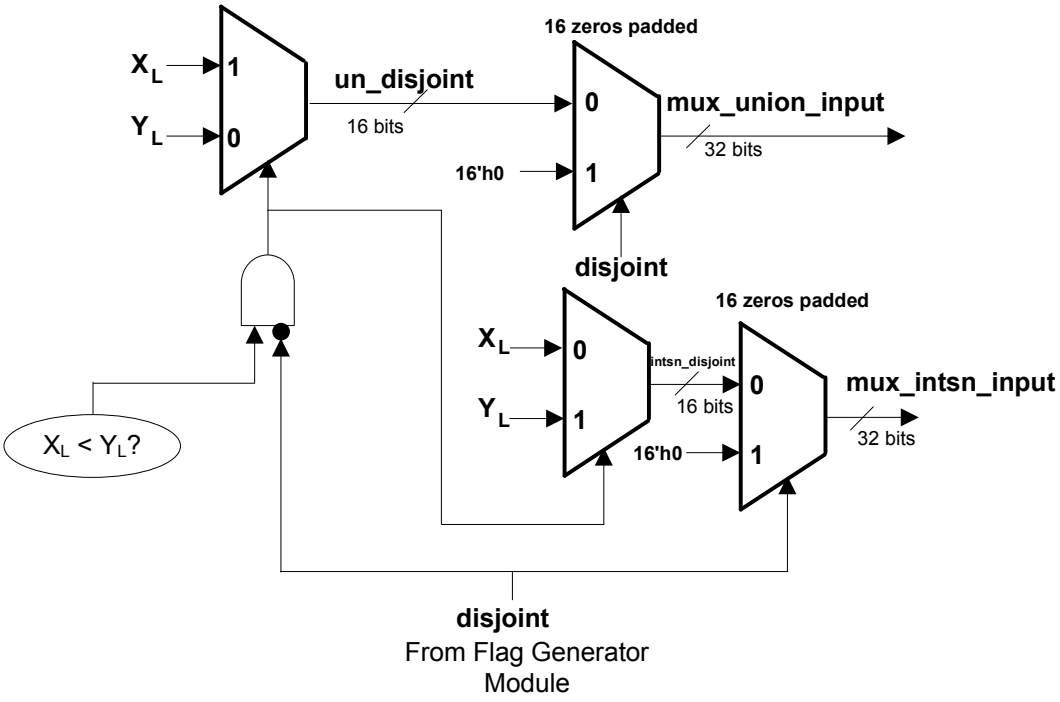


Figure 5.12: Logic for interval Union and Intersection in Lower Bound module

logic design used for this path is shown in Figure 5.13. Left shifting is performed

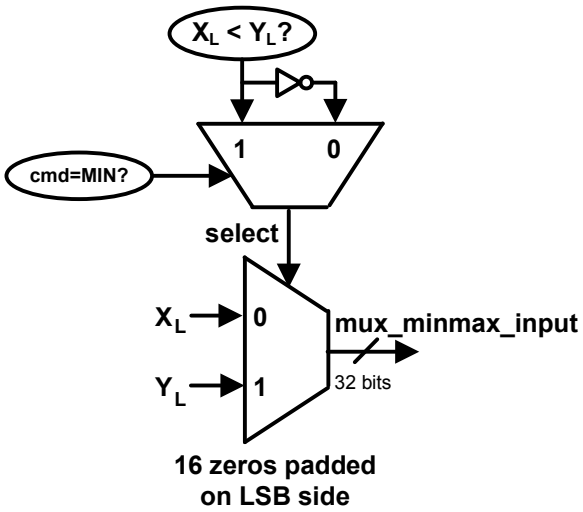


Figure 5.13: Logic for Min and Max commands in Lower Bound module

using hardware that selects out the correct value from a set of left-shifted versions of the input. The amount of shifting, in the case of interval operations, is specified by the 4-bit input signal *macexp2* for both Lower Bound and Upper Bound modules. In the case of point-wise operations, *macexp1* is given to Lower Bound module while *macexp2* is given to the upper bound module. This instruction is part of the BFP support provided by the interval ALU.

D. OR, AND, XOR

From the perspective of an ALU, the command set is complete when the ability to perform logical operations is included. Since the interval ALU has been modified to function as two independent ALUs that perform point-wise computations, logical operations such as *OR*, *AND* and *XOR* are added to enhance the capabilities of the ALU. Figure 5.14 illustrates the hardware used to perform the *XOR* operation in the Lower Bound. The operation of logical *OR* and *AND* are performed in the same way.

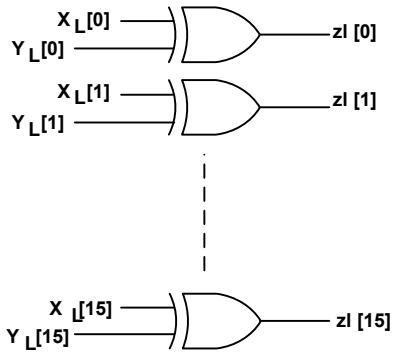


Figure 5.14: Logical XOR operation

E. Exponent Detection and Normalization

These instructions are a part of the BFP support provided in the interval ALU. They have been described in Section 4.4.2 of this work. Small-valued data is usually

sign-extended to meet large word-size requirements. For positive values, the extension comprises of 0's while for negative values, the extension comprises of 1's. Thus, the number of redundant sign bits in a number directly indicates its magnitude. 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. Table 5.7 shows the mapping of integers to the result of the XOR operation. Figure 5.15 illustrates the scheme to perform this operation.

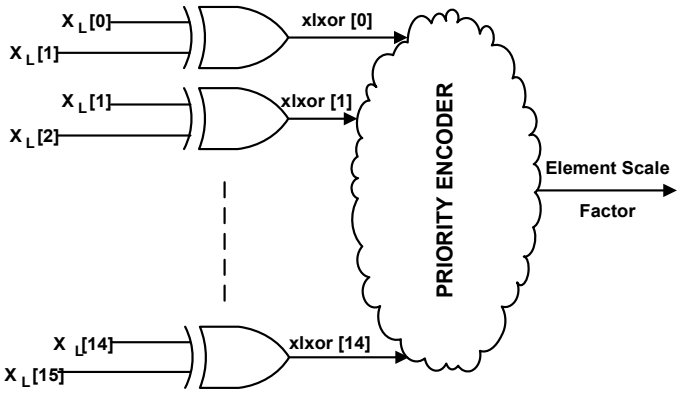


Figure 5.15: Exponent Detection scheme

Exponent Detection is a part of Normalization operation. Hence, Normalization takes the integer output of Exponent detection and on the basis of this value, it chooses the normalized value from a set of left-shifted input data elements. Figure 5.16 illustrates the hardware scheme chosen to perform this operation.

The procedure for updating the block exponent is always an integer operation. The ALU can perform the operations of block exponent increment or decrement through the addition and subtraction commands. Hence, the ALU supports the block exponent operations completely.

Table 5.7: Priority Encoder array for Exponent Detection

Priority	*Number of redundant sign bits*
1xxxxxxxxxxxxxx	0
01xxxxxxxxxxxxx	1
001xxxxxxxxxxxx	2
0001xxxxxxxxxxx	3
00001xxxxxxxxxx	4
000001xxxxxxxxx	5
0000001xxxxxxxx	6
00000001xxxxxxx	7
000000001xxxxxx	8
0000000001xxxxx	9
00000000001xxxx	A
000000000001xxx	B
0000000000001xx	C
00000000000001x	D
000000000000001	E
000000000000000	F

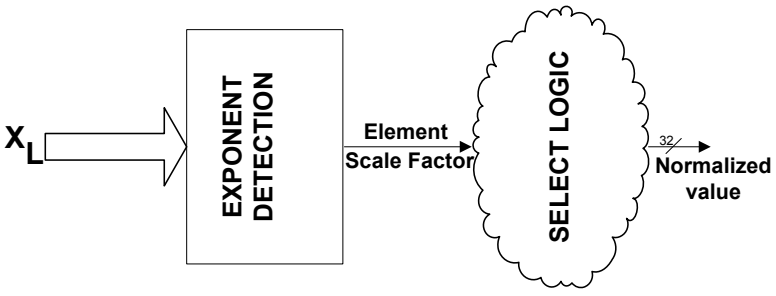


Figure 5.16: Normalization scheme

Each operation is performed and the LSB side of the results is padded with zeros so that the output of all operations is of 32-bits. The relevant result is selected out by the Select Logic block, shown in Figure 5.6, based on the present command. This final value is the output of the Lower Bound module indicated by Z_L.

The Upper Bound module is very similar to the Lower Bound module at the architectural level. The differences pertain mainly to the inputs that are operated upon for various commands. The output of this module is Z_U.

5.1.3 Scale Synchronizer module

The top-level view of the Scale Synchronizer module is shown in Figure 5.17. This module contains mostly decision logic and hence it is mostly comprised of mutiplexors.

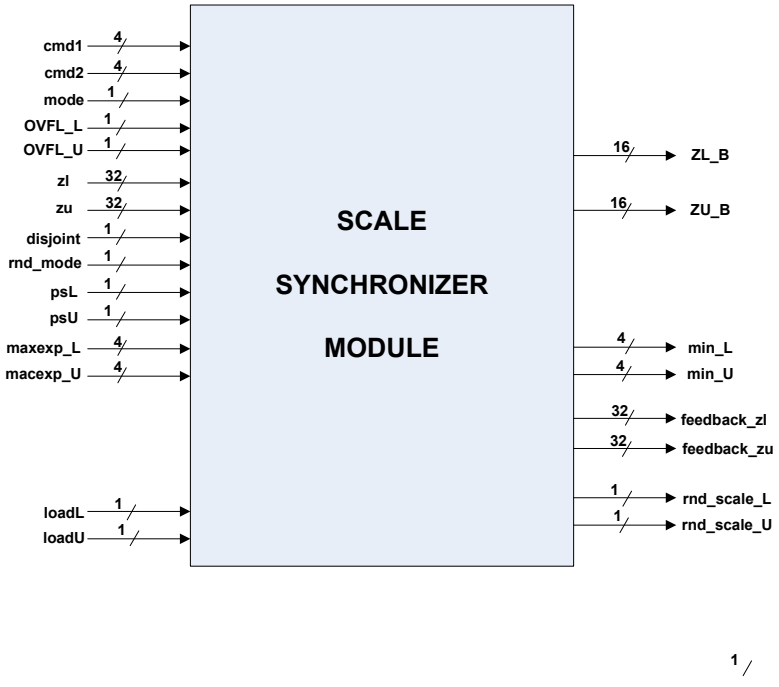


Figure 5.17: Top Level View of the Scale Synchronizer module

We first specify the inputs to this module. Input commands *cmd1*, *cmd2*, *mode*, and *rnd_mode* are delayed input signals to this module. Signals *loadL* and *loadU* are used to load new block exponent increment values. Signal *disjoint* is the delayed output of the Flag Generator module. Signals *psL* and *psU* represent delayed *permitscaling1* and *permitscaling2* respectively. Signals *maxexp_L* and *maxexp_U* represent delayed signals *maxexp1* and *maxexp2*. The 32-bit outputs of the Lower and Upper Bound modules *zl* and *zu* are fed into the Scale Synchronizer module.

The Scale Synchronizer module puts out the 16-bit output interval endpoints. These output ports are labeled ZL_B and ZU_B. The feedback to the MAC in both the Lower and Upper Bound modules is given through 16-bit output ports labeled *feedback_zl* and *feedback_zu*. This module also stores the least number of redundant sign bits while traversing through a data block. The minimum is stored in the 4-bit *min_L* and *min_U* output registers. Rounding to $+\infty$ can lead to overflow and signals *rnd_scale_L* and *rnd_scale_U* indicate this condition by going high. A detailed description of the module is provided in the ensuing lines.

A. Need for Scale Synchronization

Overflow can occur in either the Lower Bound module or in 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.

B. Main Functions of the Scale Synchronizer

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. The functions of this module are listed below:

- 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 because both output interval endpoints must be on the same scale. It also updates registers *updt_L* and *updt_U*, which store the increment in output block exponent, depending upon the status of overflow signals from the Lower Bound and Upper Bound modules.
- It stores the minimum exponent detected in a data block during Exponent Detection, so that Left shifting can follow immediately for Block Normalization.

1. The Rounding Scheme

Before discussing the procedure for synchronizing the scale of the output interval endpoints, it is important to understand the rounding scheme used in the BFPIALU. A study of the rounding modes is important because it has a direct impact on the method of applying synchronization to the intermediate interval outputs and updating the output block exponent. Rounding to $+\infty$ in either the *interval* or *pointwise* mode of operation can result in unintentional overflow since it entails the addition of an LSB to account for discarded bits beyond the output precision.

Outward rounding is the default rounding scheme for interval operations. This means that the 32-bit intermediate results from the Lower and Upper Bound modules are truncated and rounded to $+\infty$ respectively to reduce the 32-bit intermediate interval result to a 16-bit output interval. Rounding to $+\infty$ can lead to overflow errors and this affects the process of output endpoints' scale synchronization.

In this work, we have incorporated modifications in the structure of the dedicated interval ALU [9] to enable us to perform independent point-wise computations in the Lower and Upper Bound modules. In order to address the issue of the selection of the rounding scheme for point-wise computations, rounding to $+\infty$ has been extended to the Lower Bound module while truncation has been extended to the Upper Bound module. Saturation arithmetic is not implemented in this architecture since it is not suitable for intervals. This topic has been discussed in Section 4.3. The input line *rnd_mode* is asserted high to enforce truncation or rounding to $+\infty$ in the Lower and Upper Bound modules. No synchronization is required for the *point̄wise* mode of operation in the BFPIALU since both the Lower and Upper Bound modules function independently. However, updating the increment in the output block exponent requires that we be sensitive to whether rounding to $+\infty$ can lead to overflow and whether the computation is iterative.

Special Case of Rounding

As explained in section 4.3.2, rounding to $+\infty$ entails the addition of an LSB to account for the nonzero value of the discarded least significant 16-bits in the 32-bit intermediate result. When the 32-bit value is positive and in the form 7FFFXXXX (hex) with at least one X being non-zero, a '1' (hex) is added to 7FFF (hex) to perform the rounding. The resulting value is 8000 (hex) and wrap around is observed since it exceeds the highest representable positive quantity in Q0.15 fixed point format. This can lead to unreliable results if not handled properly. Such a situation can occur in the Upper Bound result in either the *interval* mode or *point̄wise* mode and in the Lower Bound result in the *point̄wise* 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 situation cannot arise with negative numbers because the addition of a positive LSB to a negative number does not meet the condition for overflow mentioned in Section 4.2.1 B.

Figure 5.18 illustrates the hardware structure to address this issue when the bit pattern appears in the 32-bit output of the Upper Bound module. The same structure is duplicated while rounding the result of the Lower Bound module.

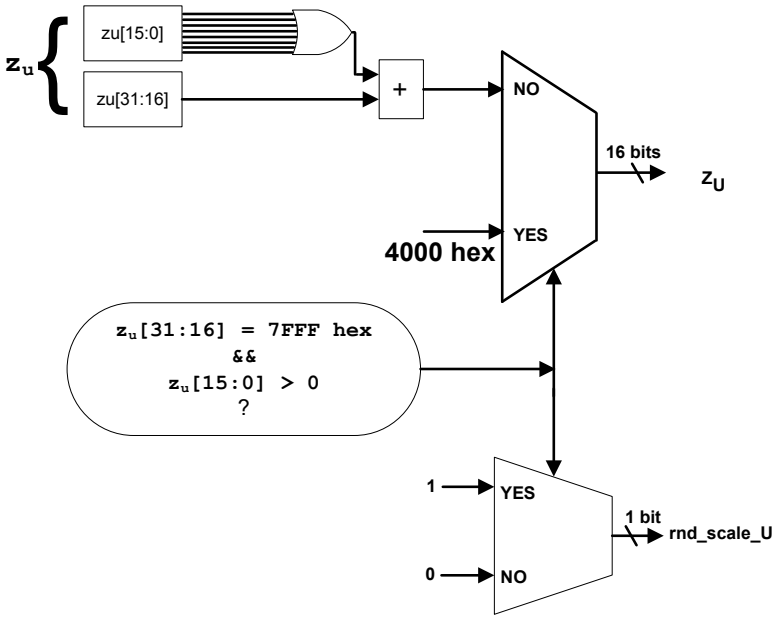


Figure 5.18: Hardware module to Handle the Special Case of Rounding

2. Synchronization and Updating the Output Block Exponent

The synchronization process depends upon the mode of operation of the BFPI-ALU in addition to overflow and rounding schemes. Updating the increment in output block exponent depends upon whether the operation is iterative or not. An iterative computation is one that involves multiple operations so that any intermediate overflows will have to be handled carefully by updating registers *updt_L* and *updt_U* and scaling the inputs for further computations. Section 3.2.3 has illustrated this clearly.

Whether a computation is iterative or not is indicated by asserting input lines *permitscaling1* and *permitscaling2* high for operations in the Lower Bound and Upper

Bound modules, respectively. For the *interval* mode of operation, *permitscaling1* is fed into both the Upper and Lower Bound modules. However, for point-wise mode of operations, *permitscaling1* is fed into the Lower Bound module and *permitscaling2* is fed into the Upper Bound module. Signals *permitscaling1* and *permitscaling2* are delayed by one by cycle and fed into the Scale Synchronizer module as *psL* and *psU*, respectively. 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 of the output block exponent increments. We discuss the process of synchronization for intervals and point-wise operations in detail next.

i) Synchronizing intervals

The synchronization of the interval output and the value of the updated registers *updt_L* and *updt_U* is shown in Table 5.8. Both endpoints share common block exponent for an interval operation and the same value is loaded in the output registers *updt_L* and *updt_U*. Care must be taken to clear these registers initially so that the computations can proceed properly.

The overflow flags from the Upper and the Lower Bound modules are represented as OVFL_U and OVFL_L respectively. They indicate the status of overflow in the computations performed in their respective modules. For an interval operation, the special case of rounding can only occur with the Upper Bound result. The signal *SplRnd* is a special flag used to identify if the bit pattern of the 32-bit value from the Upper Bound module matches 7FFFXXXX hex where XXXX is non-zero. Whenever the special case of rounding is observed, the output is fixed at 4000 hex. Depending upon the values of these flags, 16 different combinations are possible and Table 5.8 presents the actions associated with them all. This table indicates the operations to be performed for a specific combination. For instance, when the flag combination is 0100 (binary) implying that overflow occurs only in the Lower Bound module, we

recognize that the value of register zl obtained from the Lower Bound module is halved as a result of overflow correction. Table 5.8 shows ZL_B = zl instead of ZL_B = zl/2 suggesting that the result from the Lower Bound module is to be retained. On the other hand, ZU_B = zu/2 indicates that the intermediate output from the Upper Bound module is to be scaled down by 2 for this case.

A special case to be noticed is when the flag combination is 0011 and 1011 for non-iterative and iterative operations respectively. Both cases, however, essentially indicate the same scenario. In the first case, the result of the operation in the Upper Bound is halved due to overflow and then the result further qualifies for the special rounding. This calls for a scaling of factor 4 for the result of the Lower Bound which does not see any overflow. Yet another interesting case is 0110 and 1110. In both these cases, only registers *updt_L* and *updt_U* need to be updated because both the Upper and Lower Bound modules have undergone overflow.

ii) Pointwise Operations

In the case of pointwise operations, no synchronization of intermediate outputs is necessary. The focus is on updating the *updt_L* and *updt_U* registers properly given that the rounding modes can now either be truncation or rounding to $+\infty$. At any point of time, both the lower bound modules and the upper bound modules can either truncate or round the results to $+\infty$. At this point, the question arises as to why a common rounding mode (either truncation or rounding to $+\infty$) is enforced on both the Upper and Lower Bound modules through the use of a common *rnd_mode* signal. The reason for not using two input lines to enforce different rounding schemes in the Upper and Lower Bound modules is indicated in the ensuing lines.

The scale synchronizer module is essentially a large combinational logic block whose outputs are registered in flip-flops at every active clock edge. It is mainly comprised of a multiplexor that selects out a group of signals which indicate a specific set of actions performed. The selection of a particular group of signals is also based

Table 5.8: Scale Synchronization for Interval Operations

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

upon the rounding mode. Therefore, with an input line each to indicate the rounding mode for the Upper and Lower Bound module, there is more selection to be done than if a single rounding mode were enforced on both modules simultaneously.

Table 5.9 indicates the actions associated with obtaining the right output ZL_B and the updated block exponent increment value *updt_L* when the rounding mode is truncation. Similarly, Table 5.10 indicates the actions associated with obtaining the right output ZU_B and the updated block exponent increment value *updt_U* when the rounding mode is truncation for the Upper Bound module.

Table 5.9: Scale Synchronization for Lower Bound module : Truncation

psL	$OVFL\_L$	ZL_B	$updt\_L$
0	0	zl	0
0	1	zl	1
1	0	zl	updt_L + 1
1	1	zl	updt_L + 2

Table 5.10: Scale Synchronization for Upper Bound module : Truncation

psU	$OVFL\_U$	ZU_B	$updt\_U$
0	0	zu	0
0	1	zu	1
1	0	zu	updt_U + 1
1	1	zu	updt_U + 2

Table 5.11 indicates the actions associated with obtaining the output ZL_B and the updated block exponent increment value *updt_L* when we round to $+\infty$ in the Lower Bound module. Similarly, Table 5.12 indicates the actions associated with obtained the right output ZU_B and the updated block exponent increment value *updt_U* when the result of the Upper Bound module is rounded to $+\infty$.

It may be noted from Table 5.12 and Table 5.11 that the output block exponents are incremented with respect to 0 for non-iterative operations and with respect to the previous value of the block exponent for iterative operations.

The Scale Synchronizer aids the accurate computation in the results of the MAC operation by sending the *synchronized* and *unrounded* 32-bit intermediate outputs back to the Upper and Lower Bound modules. The Upper Bound module is unaware of the status of overflow in the Lower Bound module and vice versa. This could

Table 5.11: Scale Synchronization for Lower Bound module : Rounding to $+\infty$

psL	$OVFL\_L$	$SplRnd$	ZL_B	$updt\_L$
0	0	0	zl	0
0	0	1	4000 hex	1
0	1	0	zl	1
0	1	1	4000 hex	2
1	0	0	zl	updt_L
1	0	1	4000 hex	updt_L + 1
1	1	0	zl	updt_L + 1
1	1	1	4000 hex	updt_L + 2

result in erroneous MAC output if the output is fed back directly. Instead, the Scale Synchronizer synchronizes the 32-bit intermediate outputs and dispatches them to their respective modules. By not performing rounding and retaining all 32-bits, the feedback value is more accurate as opposed to dispatching the rounded value. This results in accurate MAC. The hardware scheme is clearly illustrated in Figure 5.10.

iii) Loading registers *updt_L* and *updt_U*

This design also provides the ability to load 4-bit block exponent increment values into the updt_L and updt_U registers. This is especially important when an iterative operation must be stalled mid-way in order to perform higher priority operations and then resumed. Consider the following hypothetical problem pertaining to the multiply-accumulate (MAC) operation with Q0.15 numbers steadily decreasing in magnitude. The problem is to perform the MAC operation a fixed number of times, say 100, and then check if the interval width of the last computed term fell within a predetermined value. If it did, then the operation is halted; otherwise the next 100 MAC operations are performed. In this scenario, the values of updt_L and updt_U must be stored away after a set of 100 MAC operations. Then the output block

Table 5.12: Scale Synchronization for Upper Bound module : Rounding to $+\infty$

psU	*OVFL_U*	*SplRnd*	*ZU_B*	*updt_U*
0	0	0	zu	0
0	0	1	4000 hex	1
0	1	0	zu	1
0	1	1	4000 hex	2
1	0	0	zu	updt_U
1	0	1	4000 hex	updt_U + 1
1	1	0	zu	updt_U + 1
1	1	1	4000 hex	updt_U + 2

exponent increment values must be cleared so that the inputs do not get scaled. This must be followed by a width operation for the last interval product obtained and a subtraction with the threshold fixed to perform the comparison. Assuming that the threshold is smaller in value, the MAC operations must be resumed. Since the MAC operations are iterative, the former values of updt_L and updt_U can now be restored and the computations can be resumed.

The input lines *loadL* and *loadU* along with the 4-bit signals *maxexp1* and *maxexp2* are used to perform this loading operation. The *mode* and *loadL* signals are asserted high and the block exponent increment value is applied to the 4-bit *maxexp1* input line to resume interval computations. The value of *maxexp1* is loaded into updt_L and updt_U in the next clock cycle. Signal *mode* is asserted low and signals *loadL* and *loadU* are asserted low to resume point-wise computations. The block exponent increment for the Lower Bound module is applied to the *maxexp1* input line while that for the Upper Bound module is applied to the *maxexp2* input line. Therefore, the value of *maxexp1* is loaded into updt_L and the value of *maxexp2* is loaded into updt_U. The hardware corresponding to this is a register with a multiplexor to select the value of *maxexp1* or *maxexp2* with *loadL* and *load2* as select lines.

3. Storing the minimum value of the Exponent Detected

As discussed earlier, the process of Exponent Detection results in the identification of the value of γ based on the method of finding the least number of leading sign bits. The Scale Synchronizer houses a logic block that stores the minimum exponent detected during the process of Exponent Detection on a block of data in output registers *min_L* and *min_U*. The size of these registers is only 4-bits each to account for the exponent size. This module uses a large 4-bit value (F hex) for *min_L* and *min_U* for operations other than Exponent Detection. The module successively replaces the contents of these registers with the smallest values of exponents detected so far. When the complete data block is traversed for Exponent Detection, the registers *min_L* and *min_U* are left with the least values of exponents. Thus, left shifting to normalize the block can be performed starting from the immediate next clock cycle in the case of a non-pipelined design. Incorporating such a scheme helps avoid the use of a *min* operation following each Exponent Detection operation and speeds up the BFP operations.

5.1.4 Scale_L and Scale_U modules

As mentioned earlier, overflow in the intermediate stages of iterative computations needs to be handled by scaling of the inputs so that the computations may proceed. This calls for a scaling module that shifts the input values based on the current value of the block exponent. For example, consider the evaluation of an interval summation which has undergone three overflows already. Therefore, registers *updt_L* and *updt_U* would have been incremented three times so that their value would be +3. Hence, when the next term comes in for addition, the term is scaled down by right shifting by 3 positions so that it can be added to the existing sum and the computation may proceed.

This module consists of various signed right-shifted versions of the input subjected to appropriate rounding to ensure that there is no underestimation of the input values.

Truncation is performed on the shifted inputs in the Scale_L module while rounding to $+\infty$ is performed on the shifted inputs in the Scale_U module. The high-level block diagram for this module is shown in figure 5.19.

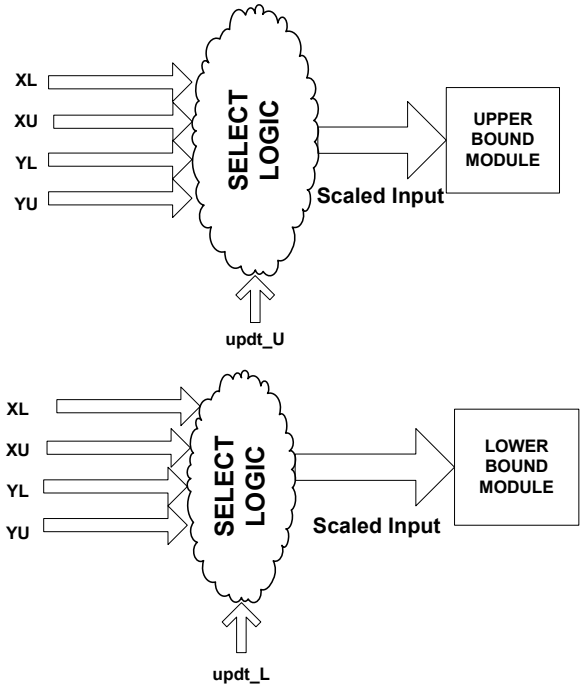


Figure 5.19: Scaling Modules for Iterative Computations

This completes the description of the hardware modules that constitute the basic non-pipelined interval ALU architecture. The throughput obtained from this architecture can be improved by pipelining the architecture. Most of the logic is found to be concentrated in the Lower Bound and Upper Bound modules of this design. Hence, pipelining is done to split up the timing in these critical modules. The following section describes this technique of improving the throughput.

5.2 Pipelined Architecture of the Design

This section describes the pipelined architecture of the design in detail. *Pipelining* is the technique of reducing the critical path in the design, thereby reducing the clock rate. It is performed by splitting the critical path of the design into smaller paths by inserting registers. Since the timing for a design is always calculated between registers, more registers inserted uniformly into the critical path imply proportionately less work to be performed between the clock edges, thereby increasing the maximum clock frequency that this design can perform reliably. Evidently, the gain of higher clock rate is obtained at the cost of higher area consumption and power dissipation. Despite its associated disadvantages, pipelining is very relevant to signal processing applications because it improves the throughput for the system.

When pipelining is performed on an already pipelined design, the process is known as Superpipelining. *Superpipelining* is the technique of dividing major stages of a *pipeline* into sub-stages [31]. It causes the work to be split into smaller logical divisions that require reduced amount of computation time. While this indicates theoretically that extremely high clock speeds can be achieved using this technique, in practice, there are timing overheads to be considered that make superpipelining ineffective beyond an optimal point.

5.2.1 Need for Pipelining

The non-pipelined architecture presented above has been synthesized in hardware. The result from the timing report of design synthesis shows that the critical path starts at input *permitscaling2* in the Flag Generator module and then passes through one of the two multipliers in the Upper Bound module. It then passes through the accumulation logic and the final selection logic for the Upper Bound module. The critical path then crosses over into the Scale Synchronizer module where it traverses some combinational logic and eventually terminates in the output register ZU_B.

Figure 5.20 illustrates the critical path.

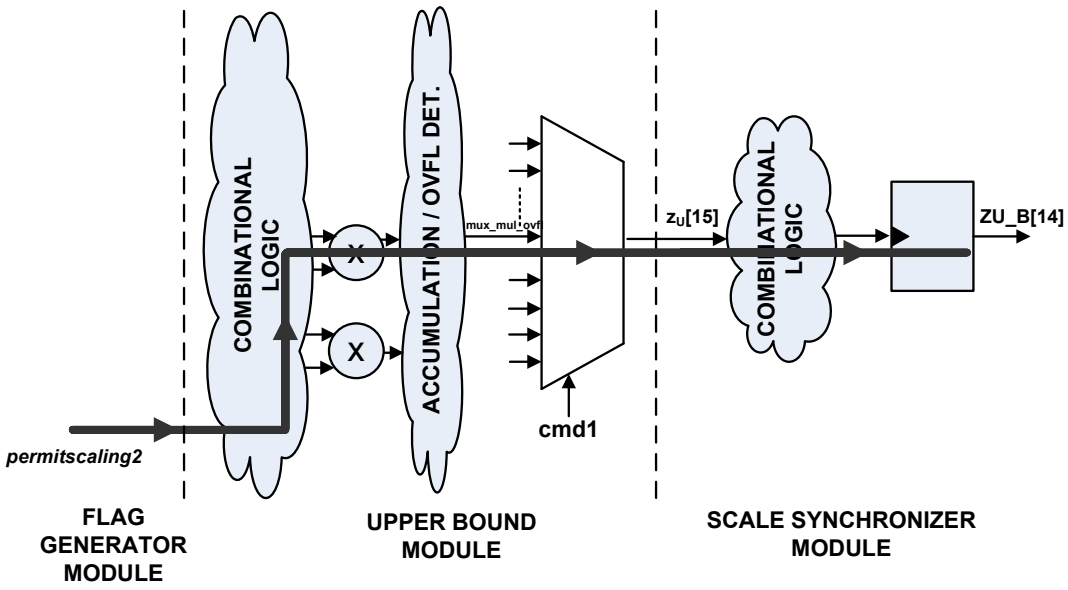


Figure 5.20: Critical path in the non-pipelined design

This long logic path must be split into smaller parts so that the clock period can be minimized. The multiplier is seen to be a prominent part of the critical path. Hence, pipelined multipliers are used to split the critical path into smaller paths.

Pipelined Multipliers for Higher Levels of Pipelining

Pipelined multipliers are based on the principle that the sum of partial products can be used to produce the final product. The Design Compiler tool from Synopsys Inc. provides a library of pipelined multiplier IPs that can be used during synthesis of the design. The multipliers based on combinational logic in the critical path of the non-pipelined architecture are replaced with pipelined multipliers to break the long critical path down into smaller parts.

The structure of the non-pipelined multiplier is as shown in Figure 5.21. With higher levels of pipelining, more partial products are stored in intermediate registers.

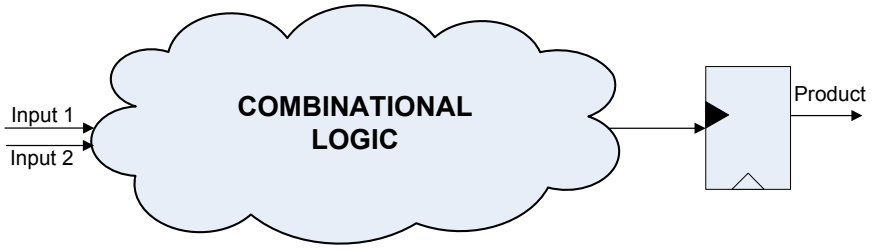


Figure 5.21: Combinational Multiplier

Figure 5.22 illustrates the abstract architecture of a three-stage pipelined multiplier while Figure 5.23 shows the same for a five-stage pipelined multiplier [9]. Due to

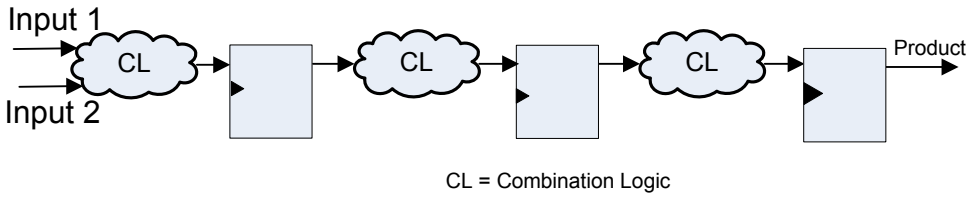


Figure 5.22: Three Stage Pipelined Multiplier

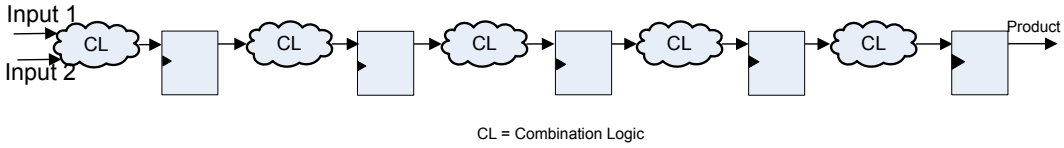


Figure 5.23: Five Stage Pipelined Multiplier

the reduced logic depth between the registers, the clock period can now be minimized further thereby yielding higher throughput.

5.2.2 Architectural Issues with Pipelined Designs

This section highlights the issues associated with pipelining the basic non-pipelined design. Pipelining decisions include whether the design should be partially pipelined

or fully pipelined and whether a pipelined MAC is a feasible solution to high clock rates. The following sections discuss these issues in greater detail.

1. Partially Pipelined Designs

A *partially pipelined* design can be obtained from the non-pipelined design by pipelining the multiplier path only. However, this can lead to collision between operations. This is illustrated in the following example.

We consider the example of a three-stage pipelined design based on a three-stage pipelined multiplier. All other operations are assumed not to be pipelined because they do not form part of the critical path. We construct a collision table that indicates the pipeline stages for the operations with time as shown in Figure 5.24. Example commands featured in it include *multiply*, *add* and *subtract*. The letter 'I' indicates the application of the input to the BFPIALU while 'O' indicates the availability of the final output. The subscript indicates the first letter of the operation being performed.

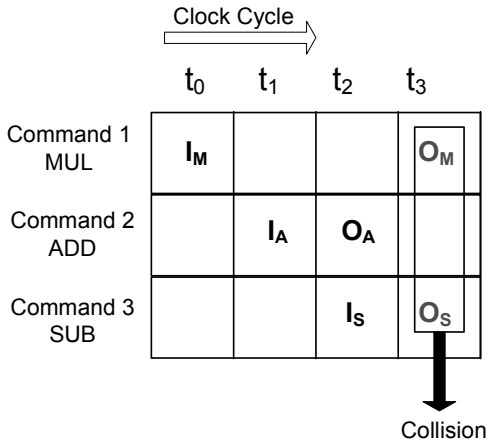


Figure 5.24: Collision Analysis for the Partially Pipelined BFPIALU

Since the multiplier is a three-stage pipelined element, its output is expected in time t_3. However, the outputs corresponding to all other operations are available

in the immediate next clock cycle because they are not pipelined. Hence, a collision occurs when *subtract* is applied at time t_2. The result of the operation is unpredictable and this is highly undesirable. Therefore, only fully pipelined designs are considered since it does not lead to collision. The collision table for a three-stage fully-pipelined design is shown in Figure 5.25. Fully-pipelined designs are obtained by delaying the inputs to the non-multiply paths by inserting registers.

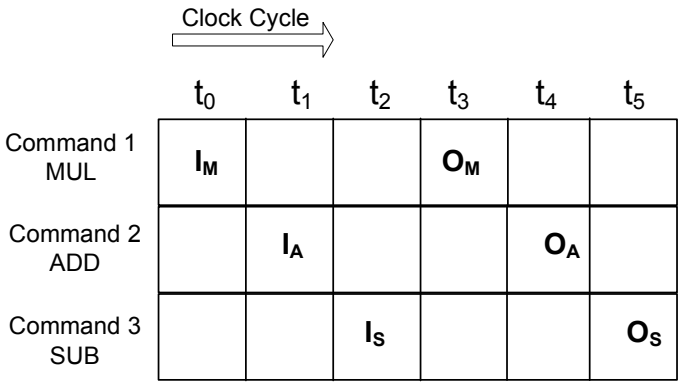


Figure 5.25: Collision Analysis for the Fully Pipelined BFPIALU

2. Pipelined MAC operation

A MAC operation uses an adder with one input being the current product and the other input being the output of the previous operation. This constitutes a feedback in the data path from the output to the adder. A multiplexor selects the output of the previous operation in the case of MAC operations and a 0 in the case of non-MAC operations for addition with the current product. The use of a pipelined multiplier does not change the dynamics of product accumulation. However, introducing delays in the feedback path creates problems of data dependency [32] because the product is available every clock cycle and there is a finite delay before the output is available in the feedback path for accumulation. The MAC will require special schemes to generate the output in this case. This also implies additional burden on the compiler

with regard to the MAC operation.

Different schemes can be used to obtain the MAC output when there is a delay in the feedback path. One such scheme is where the inputs can be fed in every alternate clock cycle. This is illustrated clearly in Figure 5.26. 'Pi' denotes the i^{th} product and N denotes the number of products to be accumulated. The example assumes N = 3. Intermediate operations used should provide a product of 0 so that the feedback remains unaffected Quite evidently, such an operation would take twice the amount

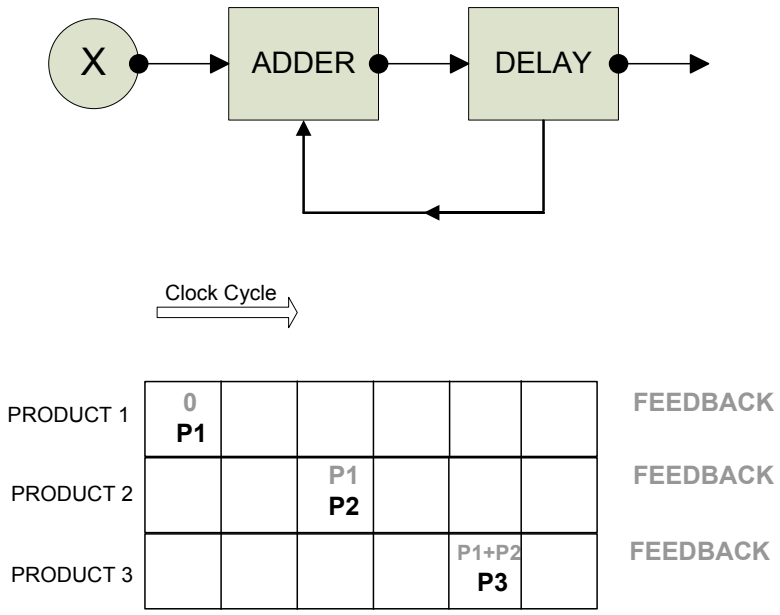


Figure 5.26: Scheme 1 : Multiply-Add pipeline

of time a normal MAC would take, even if there were no overflows. Therefore, such a scheme is not desirable for MAC operations.

Yet another scheme, though complex to implement, is described next. Figure 5.27 and Figure 5.28 illustrate this scheme for a non-pipelined design with N=7.

We feed all the inputs in order to the MAC in every clock cycle to yield product denoted by letter 'P'. This results in the feedback path circulating two different

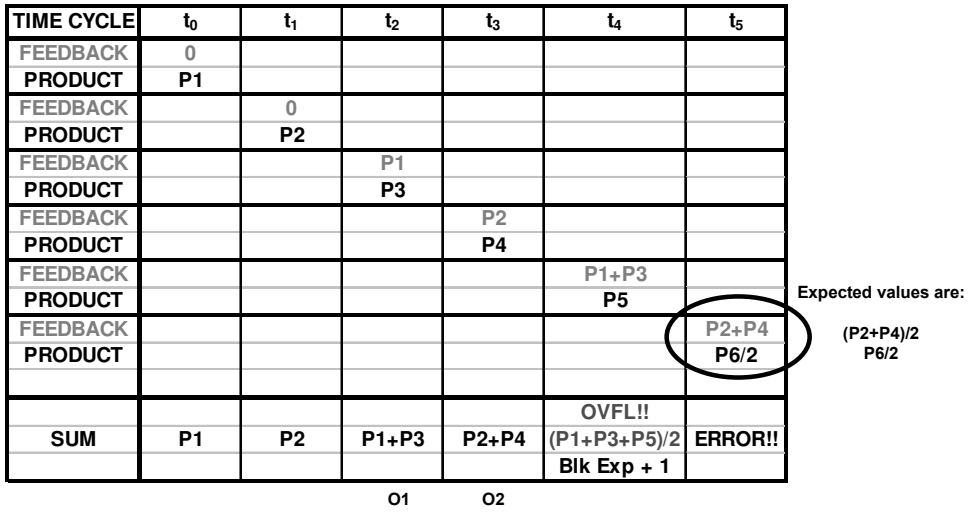


Figure 5.27: Scheme 2 : Multiply-Add pipeline

accumulated sums alternately corresponding to the accumulation of every alternate product. These can be added separately at the end of accumulation. The difficulty with the scheme pertains to the handling of overflow. This problem is highlighted in Figure 5.27. Product P6 is scaled because of the overflow. However, due to the delay in the feedback path, the input to the adder is not scaled and this leads to erroneous results. Therefore, the MAC operation must be halted at the first instance of overflow. This is detected by the increment in the output block exponent. The corrective action in this case requires the storage of MAC outputs two cycles prior to overflow. We refer to them as O1 and O2. In the example highlighted, O1 = (P1+P3) and O2 = (P2+P4). The steps enumerated below summarize the actions to resume the MAC operations:

1. Stop the MAC operation ; Clear the multiplication pipeline by multiplication of zeros.

2. Load output block exponent with post-overflow value

3. Multiply O1 with 7FFF (hex) to re-introduce it into the MAC pipeline. Keep *mac* signal asserted high during this operation.

4. Multiply O2 with 7FFF (hex) to re-introduce it into the MAC pipeline. Keep *mac* signal asserted high during this operation.

5. Resume MAC operation

The new block exponent increment value sees that the inputs into the system are scaled down by the appropriate factor. Once O1 and O2 are in the MAC pipeline, the normal course of inputs are applied and the MAC operation is resumed. The corrective operation is shown in Figure 5.28. The exact timing is important when throughput is to be evaluated. This is dealt with in Section 7.1.

TIME CYCLE					
FEEDBACK	0				
PRODUCT	O1				
FEEDBACK		0			
PRODUCT		O2			
FEEDBACK			O1/2 = (P1+P3)/2		
PRODUCT			P5/2		
FEEDBACK				O2/2 = (P2+P4)/2	
PRODUCT				P6/2	
FEEDBACK					(P1+P3+P5)/2
PRODUCT					P7/2
SUM	O1/2 = (P1+P3)/2	O2/2 = (P2+P4)/2	(P1+P3+P5)/2	(P2+P4+P6)/2	(P1+P3+P5+P7)/2

Figure 5.28: Scheme 2 : Post-Overflow Multiply-Add pipeline

This example illustrated the scheme for the non-pipelined design. For pipelined designs, however, the latency of the designs must be taken into account especially in the case of updating the input scaling factors and loading the former sums into the multiplication pipeline. Latency in pipelined designs leads to late detection of overflow. Section 7.1 discusses a scheme to handle overflows in normal MAC operations with pipelined designs.

5.2.3 Superpipelining for High Throughput

The term superpipelining may be used to describe the operation of inserting registers by observing the critical path in *pipelined* designs for levels of pipelining higher than the second stage. The second stage of pipelining for the non-pipelined architecture described above may be obtained by replacing the combinational logic based multiplier with a two-stage pipelined multiplier. This reduces the minimum clock period considerably. The three-stage pipelined design is further obtained by using a three-stage pipelined multiplier in place of the two-stage pipelined design because the critical path is found to pass through the multipliers. All the higher pipelined designs are obtained by using pipelined multipliers of greater pipeline depth. The minimum clock period falls initially and then rises after hitting a minimum value in the 7th stage of pipelining. This is attributed to the law of diminishing returns, caused by fixed timing constraints of input and output delays associated with the registers during the process of superpipelining.

The Highly Pipelined Architecture

The minimum clock period for pipelined designs generated solely through pipelined multipliers is seen to fall till the third stage of pipelining. Beyond this stage, the timing is not seen to improve much as opposed to significant increase in area and power consumption. This has been identified from synthesis runs on the RTL description of the design. Furthermore, latency increases with increased level of pipelining. Therefore, starting from the third stage of pipelining, registers are manually inserted in the observed critical path. The critical path for the three-stage design passes through the interface of the Upper Bound module and the Scale Synchronizer module with a timing of 5.10ns. The critical path in the architecture for the three-stage pipelined design is as shown in Figure 5.29. This design can to be used for applications such as filtering that rely heavily on MAC operations. This records higher throughput than the highly-pipelined design for MAC operations. The discussion for this presented in section 7.1.

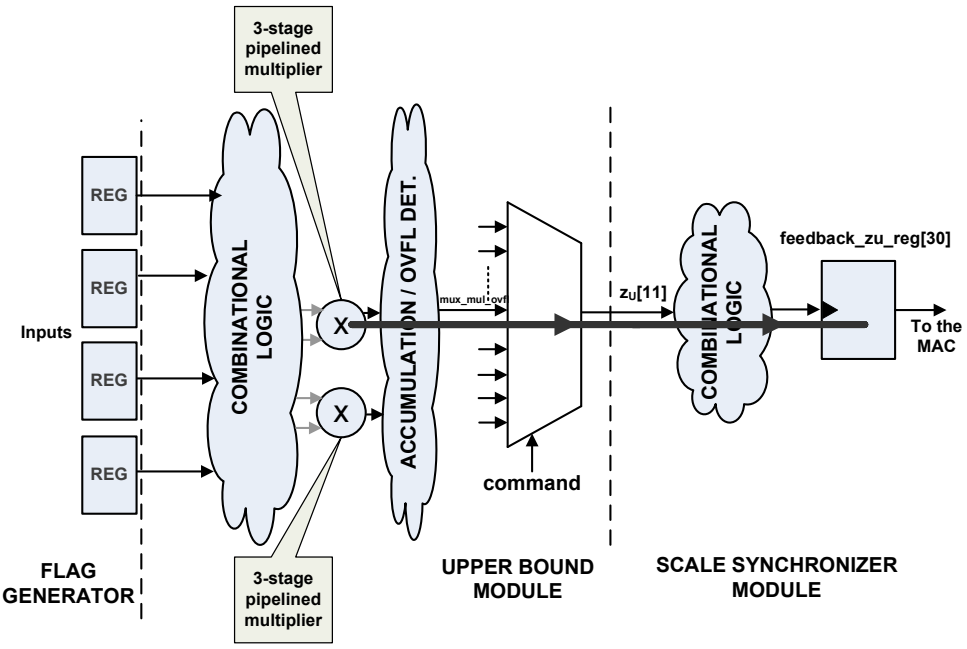


Figure 5.29: Critical path and Architecture of the Three stage Pipelined Design

Inserting a register between the boundaries of the Lower and Upper Bound modules for the three-stage pipelined design results in a *four-stage pipelined* design with significantly improved timing of 3.87ns. However, this also introduces a delay in the feedback path of the MAC structure. This design is labeled *Highly Pipelined* design.

The critical path in the architecture of the highly-pipelined design is shown in Figure 5.30. This utilizes the three stage pipelined multiplier and two registers at the boundaries of the Lower and Upper Bound modules with the Scale Synchronizer module each. This design records an improvement of 31.8% over the three-stage pipelined design and 166.9% in timing over the non-pipelined design. For MAC operations in the highly-pipelined design, schemes such as the one described above must be used. Further results pertaining to area utilization and power consumption are presented in Chapter 6.

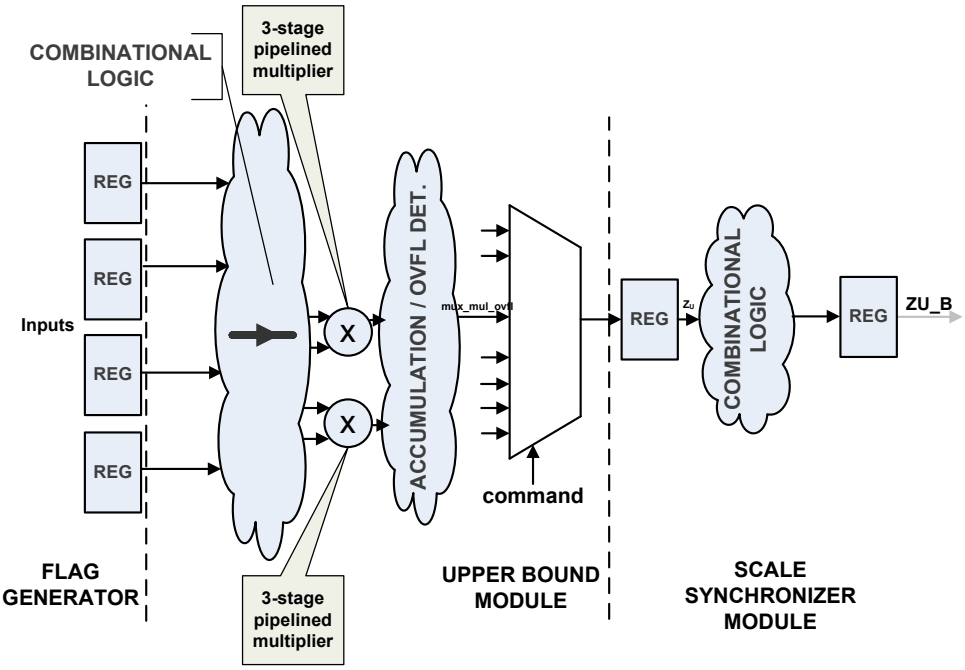


Figure 5.30: Critical path and Architecture of the Highly-Pipelined Design

Chapter 6

Testing and Results

This chapter presents the results of simulation and synthesis in terms of timing, area and power dissipation for different pipelined architectures of the interval ALU discussed in Chapter 5.

Functional verification is performed first and then the results of synthesis and power analysis are presented. This chapter aims to provide the results of synthesis and power analysis on a design that is shown to be working correctly through simulations. The first step in testing is to verify the correctness of the design functionality. Simulations are conducted and the resulting waveforms are studied. Once the simulations establish the correct functionality of the designed hardware, the design is synthesized and the area, timing and power dissipation values associated with it are noted.

6.1 Simulation Results

The simulations were conducted using test vectors in Cadence 2004 version environment. The behavior was captured in Verilog. The tests were performed for the

ALU operating in both *interval* mode and *pointwise* mode. Complete code coverage was measured by inserting *display* routines in each line. In this section, we verify the correctness of the design functionality. The simulation results presented pertain to the non-pipelined architecture. The output of a particular pipelined design is simply these waveforms delayed by the appropriate number of clock cycles to account for the associated latency. The output values remain the same.

6.1.1 Interval mode of operation

Figure 6.1 illustrates Block Normalization for a block size of 5. First, Exponent Detection is performed. The minimum value is observed in signal min_L which latches on to the smallest integer. The least number of redundant leading sign bits is identified to be 2. After the full block has been traversed, all the data samples are passed through the BFPIALU once again to left shift all samples by 2 positions. The normalized values are observed and found to be correct. Therefore, fixed point computations may be performed on this data.

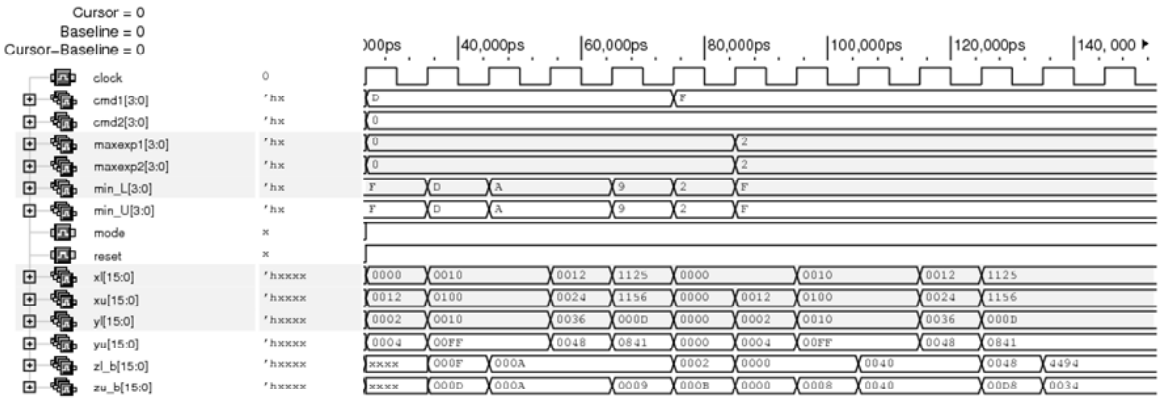


Figure 6.1: Simulation results for BFP commands (*interval* mode)

Figure 6.2 illustrates the results of the simulation for MIN, MAX and logical operations when the ALU is in the *interval mode*. Command *cmd1* is supplied to

both Lower and Upper Bound modules and hence, irrespective of command *cmd2*, *cmd1* is executed in the Upper Bound module. An important point to be noted here is that these operations do not lead to overflow.

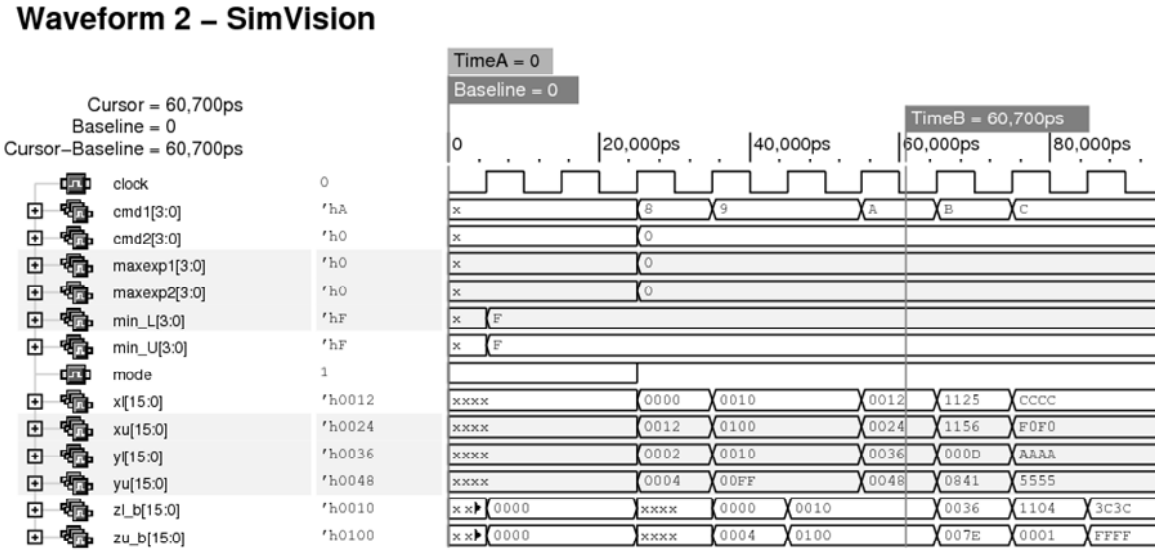


Figure 6.2: Simulation results for Min, Max, OR, AND, XOR (*interval* mode)

Figure 6.3 illustrates the handling of overflow with CBFS. It is seen that after the first addition in the iteration, the output is scaled down by a factor of 2 since overflow occurs. Since *permitscaling1* is high, it implies that the iterative operation of summing is still in progress and hence all incoming data will be scaled down by a factor of 2. The incremental block exponent is seen in output signals *updt_L1* and *updt_U1* which bear the latest values for the output. It may be noted that signals *updt_L1* and *updt_U1* are aliases for *updt_L* and *updt_U* during the simulation runs. The scaling is halted once the iteration has been completed which is signified by *permitscaling1* asserted low. The results are verified to be correct. Additions performed as part of non-iterative operations are found to result in a block exponent increment of 1. This is also illustrated in Figure 6.3.

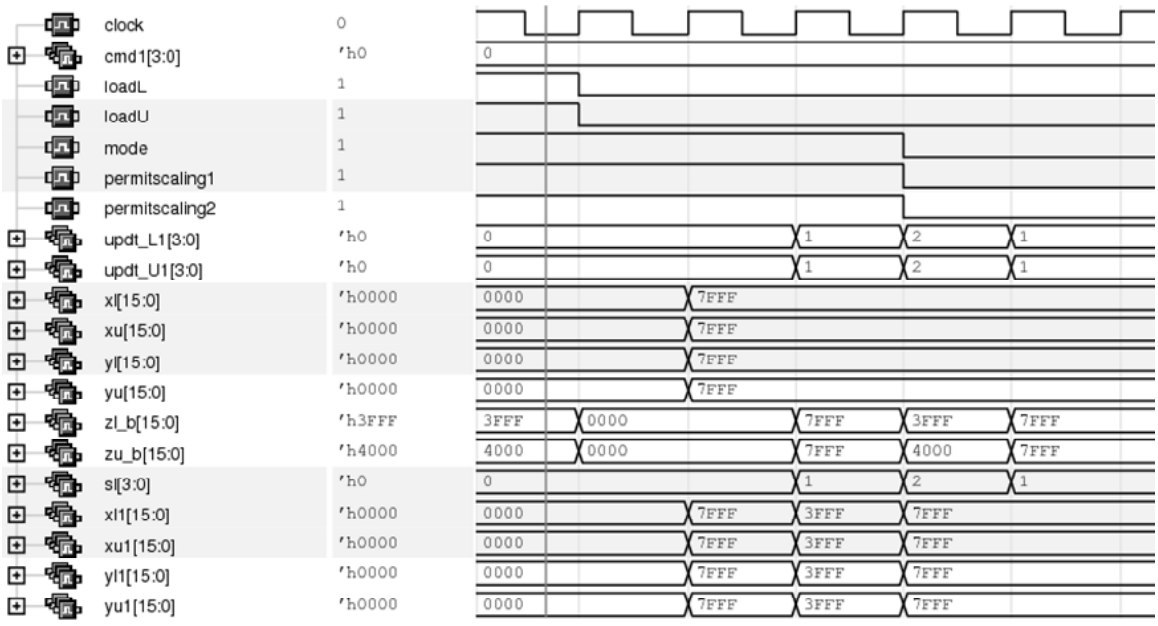


Figure 6.3: Iteration of Summing a Series with Overflow (*interval* mode)

6.1.2 Pointwise mode of operation

It is evident that in the interval mode of operation, *cmd1* is supplied to both the Lower and Upper Bound modules. However, in the *pointwise* mode of operation, *cmd2* is supplied to the Upper Bound module instead of *cmd1*. This is clear from the simulation runs presented in Figure 6.4. As seen in the figure, the rounding performed depends on the status of *rnd_mode*. Hence the same operation of multiplication yields 0001 (hex) when the result is rounded to $+\infty$ and 0000 (hex) when the result is truncated. The pertinent outputs are encircled in Figure 6.4. Figure 6.5 highlights the case of overflow in *pointwise* operations. The upper bound faces overflow before the iteration is over and hence its input is scaled. In contrast, the lower bound does not overflow and its block exponent does not get incremented. This is evident from the states of signals *updt_L1* and *updt_U1*.

The special case of rounding for the BFPIALU in the *pointwise* mode of operation is illustrated in Figure 6.6. Here, a MAC operation is carried out over a large number

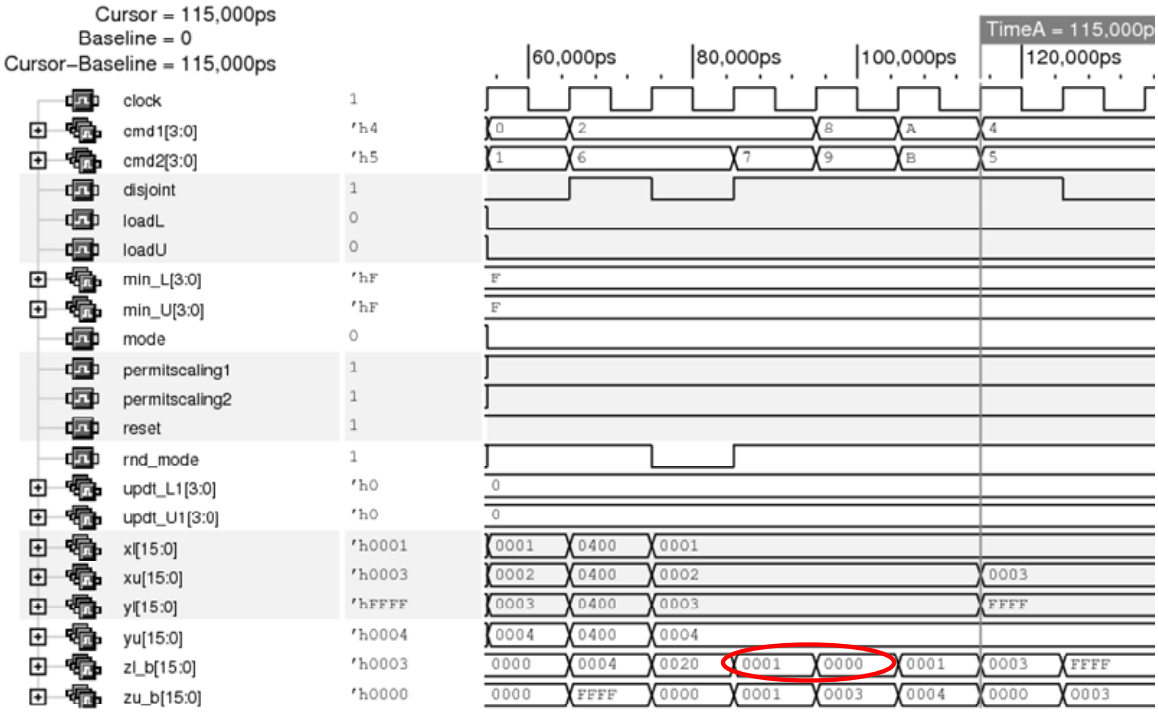


Figure 6.4: Independent pointwise operation in the interval ALU

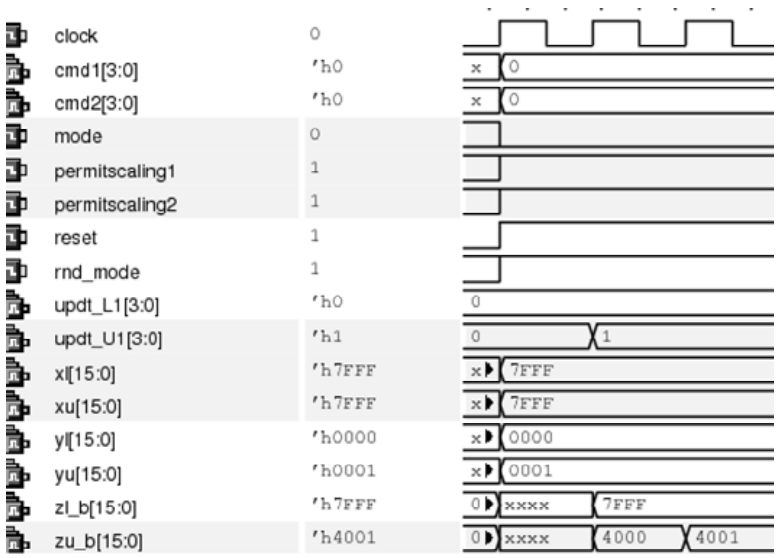


Figure 6.5: CBFS in an iteration with overflow in the Upper Bound

of computations to observe the special case of rounding. Rounding to $+\infty$ is enforced in both ALUs to observe this. The isolation between the functioning of the Lower and Upper Bound modules is evident from Figure 6.6.

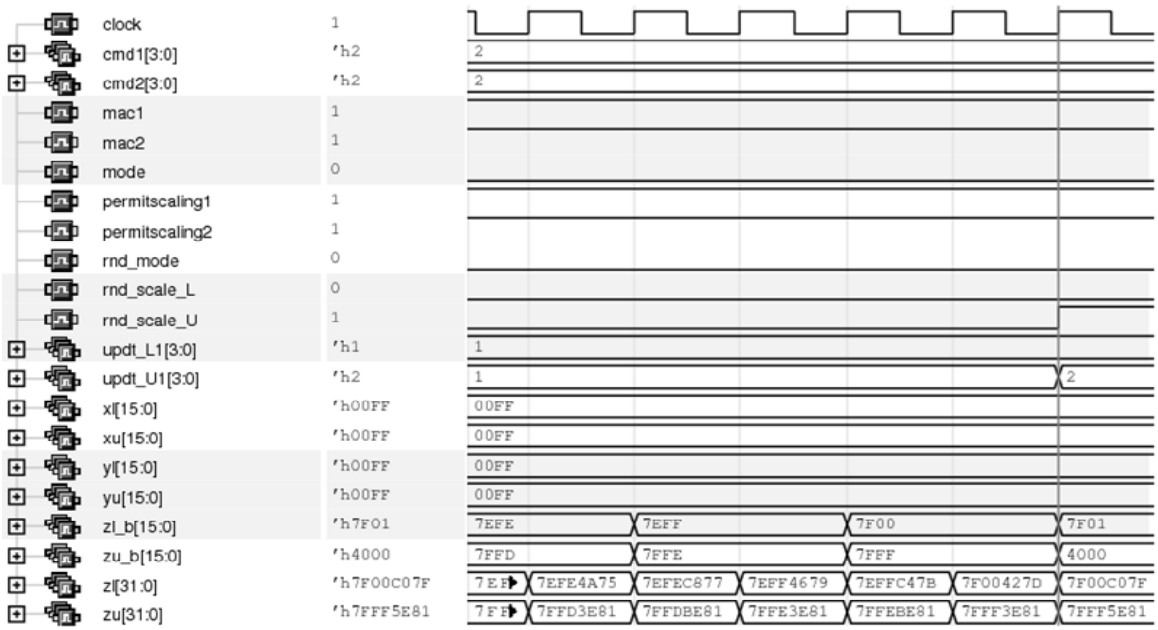


Figure 6.6: Special case of Rounding in the *pointwise* mode

6.2 Synthesis Results

In this section, the various pipelined designs discussed in the Chapter 5 are synthesized using the 0.18μm standard cell library from Oaklahoma State University [33]. Synopsys Design Compiler [34] is used to perform the timing analysis for this design while Synopsys Prime Power [35] is used to perform the power analysis. The organization of the results is as follows: First the results of synthesis for the non-pipelined design are presented. Details of the individual modules of the architecture are mentioned. Then pipelined multipliers are synthesized in order to study their timing

and area utilization. Then the results of synthesizing all pipelined architectures are presented with area and timing.

6.2.1 The Non-pipelined Architecture

Table 6.1 shows the results of synthesis for the non-pipelined design. It highlights the timing and area for individual blocks that comprise the proposed architecture. The Scale_L and Scale_U modules are included with the Lower Bound module and the Upper Bound module respectively. Figure 6.7 illustrates the area occupied by each

Table 6.1: Timing for Non-pipelined Design

Module	*Least Clock Period* (ns)	*Area* (μm^2)
Flag Generator	2.49	15425
Lower Bound	9.56	198596
Upper Bound	9.76	223646
Scale Synchronizer	3.29	39500
Overall Architecture	10.33	473703

module in the design. The greatest concentration of logic is present in the Lower and Upper Bound modules. This explains the higher area utilization and longer critical paths in these modules. The overall non-pipelined design is synthesized for a timing of 10.33ns and an area of $473703 \mu m^2$.

6.2.2 Synthesis of Pipelined Designs

Pipelined multipliers with a pipeline depth of up to 10 stages were used to obtain the pipelined designs. The results of synthesizing the pipelined designs are presented

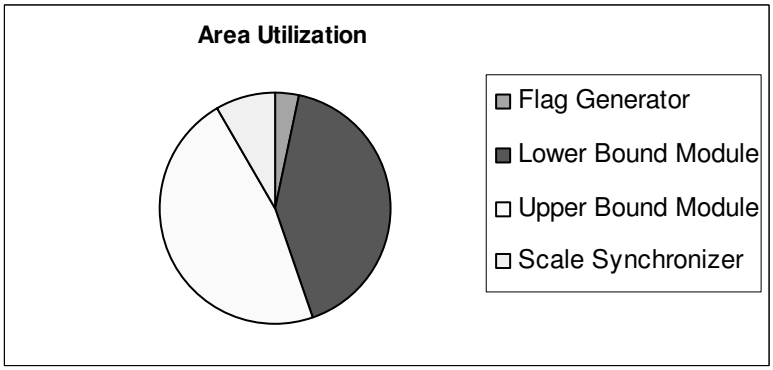


Figure 6.7: Area distribution between Modules

in Table 6.2. It shows the minimum clock period and area for every stage. The trend for minimum clock period across various superpipelined designs is captured in Figure 6.8. The least clock period is a value of 4.93ns in the seventh stage of pipelining. Beyond the third stage of pipelining, no significant reduction in clock period is observed and in fact, after the minimum value is obtained in the seventh stage, it begins to increase due to timing overheads in the design. The trend for area captured in Figure 6.9 is seen to increase with higher levels of pipelining. The growth however, with deeper pipelining, is more with respect to the work of [9] because this architecture utilizes four multipliers in all, two each in the Lower and Upper Bound modules respectively to perform interval multiplication. Furthermore, this design has more commands with a proportionally greater number of parallel paths in it. Since each pipelined design is fully pipelined, each parallel path is delayed appropriately by inserting registers with each level of superpipelining and this results in an explosion in the area utilization.

The area and timing values for the highly-pipelined design described in Section 5.2.3 is presented in Table 6.3.

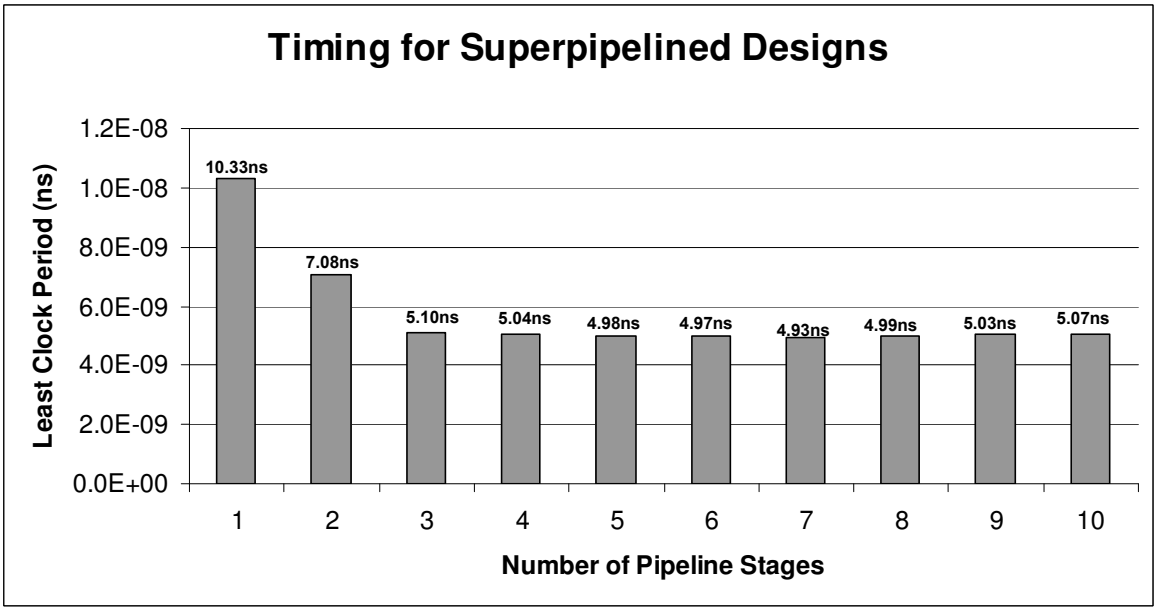


Figure 6.8: Timing report for Superpipelined designs

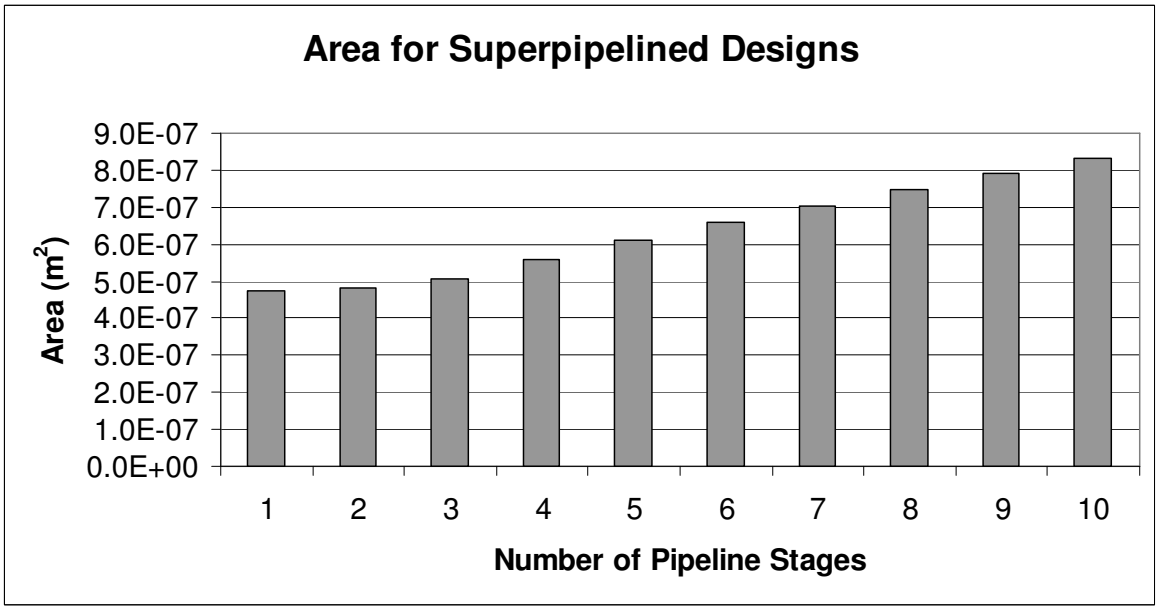


Figure 6.9: Area report for Superpipelined designs

Table 6.2: Area and Timing Report for different Superpipelined designs

Stage of Pipelining	*Least Timing (ns)*	*Area (μm^2)*
Nonpipelined	10.33	473703
2-stage	7.08	483855
3-stage	5.10	506727
4-stage	5.04	557637
5-stage	4.98	609097
6-stage	4.97	657818
7-stage	4.93	701750
8-stage	4.99	747659
9-stage	5.03	792890
10-stage	5.07	832750

Table 6.3: Area and Timing Report for the Highly-Pipelined Design

Highly − Pipelined Design	*Least clock period (ns)*	*Area (μ m^2)*
Four-stage	3.87	527184

6.3 Power Analysis

Power consumption has become an important issue in recent times owing to the high demand for battery powered devices such as mobile phones, Personal Digital Assistants (PDA), etc. A large number of VLSI designers treat power dissipation as an important metric while designing systems for such applications. In this section, random input vectors are passed through the superpipelined designs and the dynamic

power consumption is calculated using Synopsys PrimePower. PrimePower is a gate level power analysis tool which can model and evaluate dynamic power dissipation in multi-million gate designs. Its in-built algorithms consider cell state dependency, multiple loads, partial swings and nonlinear dynamic ramp effects. They also take into account glitches, multiple transitions, 'unknown' and high-impedance states [35]

The design is first synthesized using the Synopsys Design Compiler in order to perform the power analysis. A test bench with random input vectors is written for the netlist of the design generated during the synthesis. The input vectors are then driven through the netlist and a *Value Change Dump* (VCD) file is obtained. The VCD file is a dump of all signal transitions that occurred during the simulation and affects the power dissipation directly. PrimePower is then used to compute the power dissipation based on the VCD file and the synthesized design.

The SSHAFT tool [36] automates the process of evaluating power dissipation. A large number of random test input vectors is necessary to obtain a good estimate of the average power dissipation. All architectures were subjected to a power analysis with 1000 random input test vectors. The test bench was written in C. Fixed character streams corresponding to the header information, top-level module signal declarations and top-level instance creation were used to write fixed content to the output file. The random cases were generated using the *rand()* function in C. Care was taken to ensure that the upper endpoint of the intervals generated was greater in value than the lower endpoint because the interval is an ordered set of real numbers. Figure 6.10 shows the code segment used to generate the random intervals. Table 6.4 shows the power dissipation across various superpipelined architectures.

The trend for power dissipation across various superpipelined architectures is evident in Figure 6.11. With an increase in the number of pipelined stages, the power dissipation also increases. This is because the area increases proportionately with the number of pipelined stages and hence more bit toggles are encountered. The highly-pipelined design consumes an average power of 0.1247 W for 1000 input vectors.

```
xl = rand()%65535;

if(xl <= 32767)
{
      do
      {
            xu = rand()%65535;
      }while( !((xu >= xl) && (xu <= 32767)));
}
else
{
      do
      {
            xu = rand()%65535;
      }while(!((xu >= xl && xu <= 65535) || (xu >= 0 && xu <= 32767)));
}
```

Figure 6.10: Code Segment to Generate Random Intervals

Table 6.4: Power Dissipation for Pipelined Designs with 1000 input vectors

Stage of Pipelining	*Power Dissipation* (W)
Nonpipelined	0.04918
2-stage	0.06978
3-stage	0.09033
4-stage	0.1032
5-stage	0.1166
6-stage	0.1300
7-stage	0.1418
8-stage	0.1528
9-stage	0.1625
10-stage	0.1697

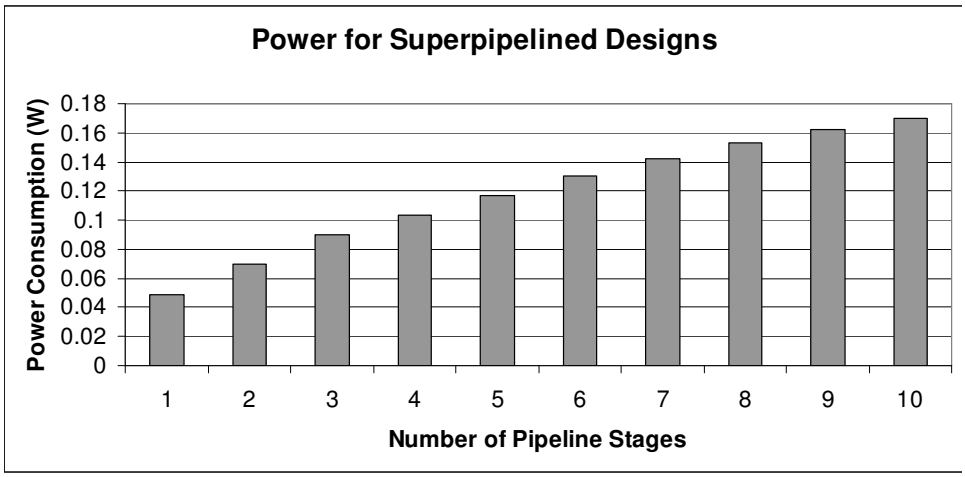


Figure 6.11: Power Dissipation for the Superpipelined designs with 1000 vectors

Chapter 7

Architectural Insights

This chapter evaluates the expression for *throughput* in a pipelined design with a known number of overflows and pipelined stages. It also describes a structural hazard associated with point-wise computations in the BFPIALU and concludes with a discussion of the impact of Input Scaling and CBFS techniques on accuracy of the output.

7.1 Throughput for the BFPIALU

7.1.1 The Probability of Overflow in the BFPIALU

A study of the statistics of overflow is important while evaluating throughput for the BFPIALU. The latency of a pipelined design and the number of overflows can vary the throughput of the BFPIALU directly. Following each overflow, we scale the inputs from that point forward down by an additional factor of 2 for the computations to come. This is equivalent to doubling the dynamic range upon each overflow and this reduces the probability for the next overflow to occur. In this section, we describe a Monte Carlo simulation to determine the frequency of overflows for MAC operations

performed in the BFPIALU.

We generated 20,000 datasets containing 4096 uniformly distributed samples each in the range [-1, 1-2^{-15}]. From the perspective of an ALU, the Uniform Distribution is the most appropriate distribution that can be assumed on the input data. We operated on two data sets at one time; the procedure is described next. Individual data samples were pulled out of two unique data sets at a time, multiplied and the product was accumulated. Each product was accumulated and upon overflow, the sum and the inputs to the operation that point forward were scaled down by a factor of 2. After 4096 accumulations, we moved on to a different pair of data sets. Counters, unique to each such dataset combination, were used to keep count of the number of overflows. At each overflow, the corresponding counter was incremented. We are aware of the fact that by scaling the inputs at each overflow, we reduce the probability of the occurrence of the next overflow. Therefore, we expect to see a falling trend in the frequency of higher number of overflows.

It was seen through simulation that at least one overflow occurred 10,000 times, at least 2 overflows occurred 9,999 times, at least 3 overflows occurred 4502 times and at least 4 overflows occurred once. We clearly see that the higher the number of overflows is, the lower its frequency of occurrence is. We utilize this falling trend to ignore the computation-cycle penalty associated with overflows as the number of computations approaches infinity while evaluating throughput in the next section.

7.1.2 Evaluating Throughput for the BFPIALU

Throughput for an interval ALU is ideally defined as the number of interval operations performed per second [9]. While this provides a good measure for the efficiency for the BFPIALU, practical usage demands that throughput be quantified in a more useful way. This is especially true with a BFP implementation because throughput 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

evaluate throughput for interval operations performed in the BFPIALU.

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

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

We assume a data block comprised of 'N' interval data samples to evaluate the throughput for interval operations. Let the architecture to be evaluated for throughput contain 'k' pipeline stages. Let 'p' be the number of overflows faced in the computation for each block. We evaluate the expression that denotes the time to process all the N input samples. In addition to the number of overflows, number of pipeline stages and the number of input samples to be processed, throughput also depends upon the type of computation to be performed.

When overflow occurs in a design, the inputs that point forward should be scaled down appropriately so that the computation can continue. However, very highly pipelined designs have high latency and this leads to late updating of the input scale factors. Therefore, some computations must be performed again and this leads to higher cycle count. Specifically, operations of *multiply-accumulate*, *addition* and *subtraction* face this situation. Multiply-accumulate involves a MAC feedback path and reintroducing the pre-overflow terms in the MAC pipeline consumes additional clock cycles. It is referred to as a Type-1 operation. However, addition and subtraction do not involve feedback in their data path. Successive addition or subtraction of independent terms is not important from the standpoint of signal processing applications. Furthermore, applications such as filtering rely on MAC operations to evaluate their output terms of a difference equation, usually rely on MAC operations for the same. Therefore, we do not analyze addition and subtraction for throughput when overflow occurs. Instead, we combine them with operations such as *division, union, intersection, midpoint, min, max, OR, AND, XOR, exponent detection, left shifting* and *direct*

normalization and analyze them when no overflow occurs. The operation of *width* is also put into the same category. We refer to this category of operations as Type-2 operations. We next analyze throughput for these two categories of operations.

7.1.3 Type-1: MAC operations in Pipelined Designs

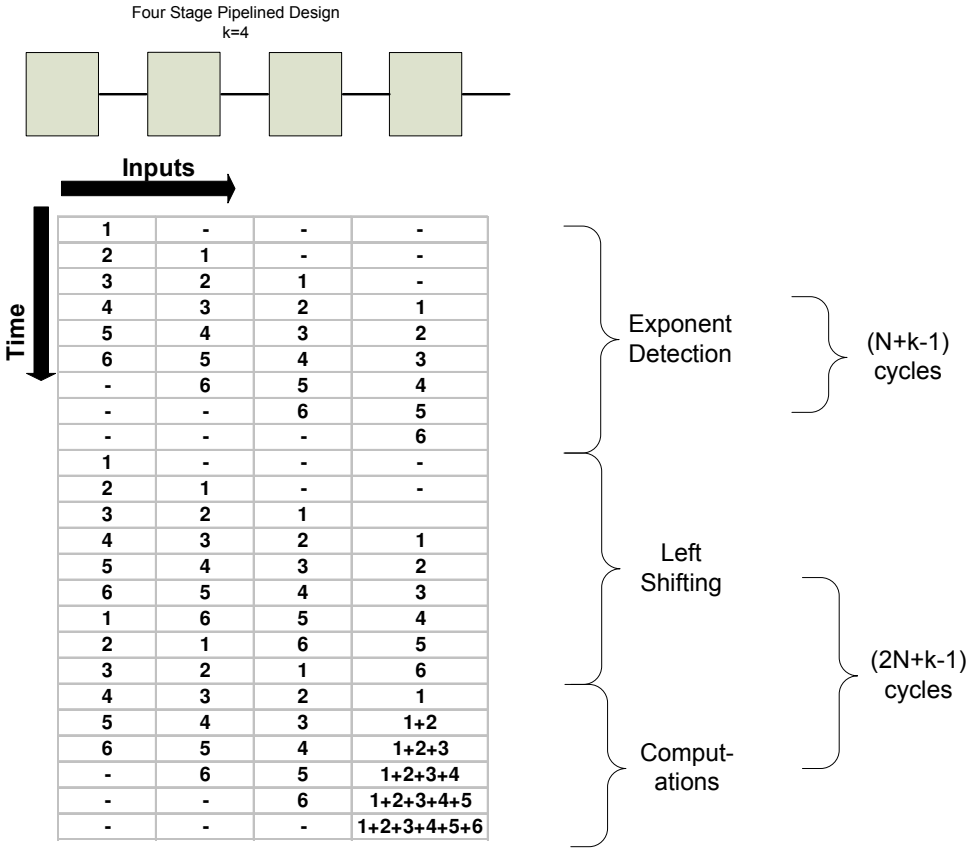
In this section, we assume that all the computations to be performed on the input samples of the data block are interval *multiply-accumulate* operations. The analysis is performed based on the assumption that we accumulate 'N' products as part of the computation. The number of clock cycles to accumulate the products is different depending upon whether or not there is a delay in the MAC feedback path. Therefore, we analyze these two cases separately. Section 7.1.3 A analyzes throughput for pipelined designs that do not involve delay in their feedback paths. This applies to the MAC operations performed in the fully-pipelined designs formed using pipelined multipliers as described in Chapter 5. Section 7.1.3 B analyzes the throughput for a MAC operation that involves a delay in its feedback path. This helps us estimate the throughput for the highly-pipelined design.

A. MAC with no delay in the Feedback Path

We compute the total number of clock cycles required to perform Exponent Detection, Left shifting and computations on the given data block. We require two full traversals through the data block to perform normalization on the data block - once to perform Exponent Detection and once again to left shift all data samples in it by a factor of γ. Thus, for a data block of size N, we need (N+k-1) cycles for Exponent Detection and (N+k-1) cycles again for left-shifting the samples.

We need not wait until the last data sample has been left-shifted to apply the input data to perform computations with the samples. Instead, we start applying the input samples immediately after applying the last data sample for left shifting. We consider the example of a system with number of interval products N=6 and pipeline

stages k=4 with no overflows as shown in Figure 7.1. Here each input numbered 1 to 6 is assumed to be a pair of numbers that generates a product for the process of accumulation. Since the MAC operation takes only one cycle, we find that the process of left-shifting the data samples and performing computations take (2N+k-1) clock cycles together.



1	-	-	-
2	1	-	-
3	2	1	-
4	3	2	1
5	4	3	2
6	5	4	3
-	6	5	4
-	-	6	5
-	-	-	6
1	-	-	-
2	1	-	-
3	2	1	
4	3	2	1
5	4	3	2
6	5	4	3
1	6	5	4
2	1	6	5
3	2	1	6
4	3	2	1
5	4	3	1+2
6	5	4	1+2+3
-	6	5	1+2+3+4
-	-	6	1+2+3+4+5
-	-	-	1+2+3+4+5+6

Figure 7.1: Example to illustrate MAC computations with N=6 and k=4

In practical situations such as performing accumulation, it is not possible to know in advance whether overflow will occur or not. When overflow occurs, the output block exponent increments over its former value. However, the problem is that this increment will be known *only after the latency period* for the k-stage pipelined design has passed. Hence, the best strategy is to keep applying all the samples assuming no overflow occurs.

Figure 7.3 illustrates the case of no-overflow and one overflow in MAC operations. The outputs are represented with letter 'O' and a number 'i' indicating that it is the output corresponding to input 'i'. For the scheme depicted in the example, the corrective action involves the following steps:

- Clear the MAC pipeline by multiplying between 0s in the clock cycle after the overflow as a non-MAC operation
- Ignore the (k+1) outputs after overflow because they correspond to the pre-overflow block exponent
- Loading the new block exponent increment value
- Re-introduce overflowed interval output into the pipeline by multiplying it with 7FFF hex. Reapply the samples after those inputs that resulted in overflow

The sequence of operations post-overflow is indicated in Figure 7.2.

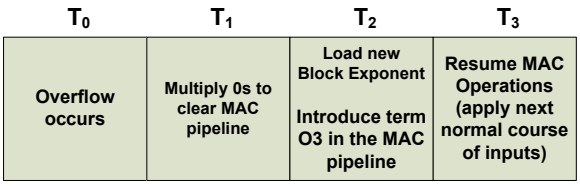


Figure 7.2: Post-overflow Corrective Actions

The output block exponent increment is loaded so that the inputs that point forward are scaled by this factor. Therefore, reliable results are obtained. For every overflows, the penalty to be borne is (k+2) clock cycles. This is clearly illustrated in Figure 7.3 which depicts the case of one overflow occurring. The times T_0, T_1, T_2 and T_3 from Figure 7.3 are to be matched with that in Figure 7.2.

The computation time is [(N+k-1)+p·(k+2)]. The time to process one block completely, inclusive of normalization, is { (N+k-1) + [(2N+k-1)+ p·(k+2)]}. This evaluates to [3N + 2(k-1) + p(k+2)] clock cycles per block. This is verified in Figure

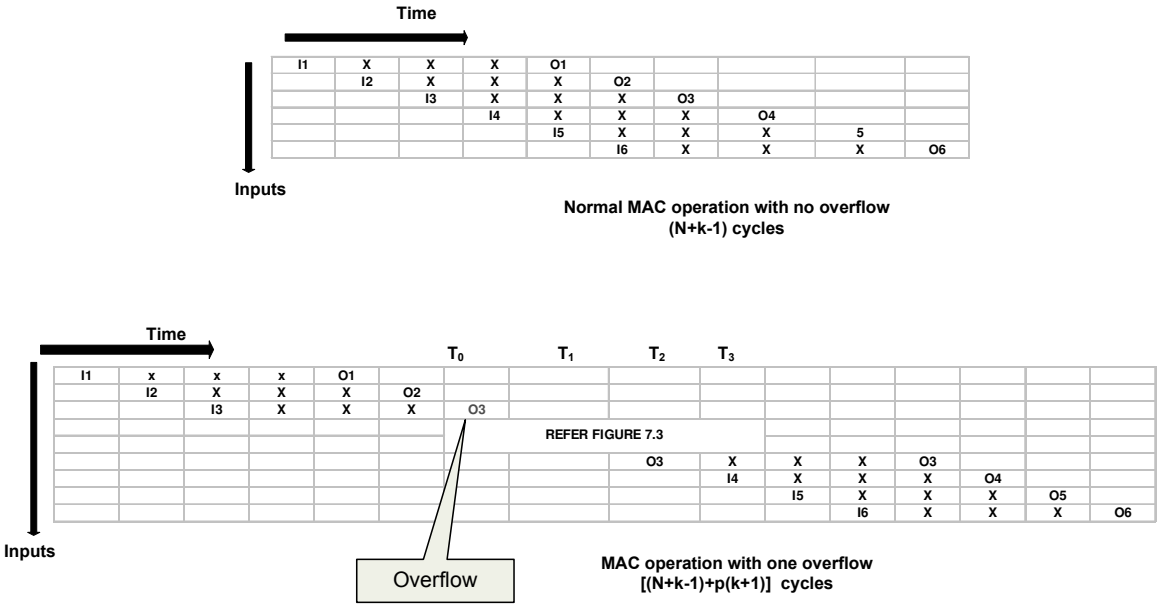


Figure 7.3: Illustration of MAC computations with one overflow for N=6 and k=4

7.3 where for N=6, k=4 and p=1, only the computation takes 15 clock cycles for one overflow as opposed to 9 clock cycles when no overflow occurs.

Denoting the least clock period at which the hardware performs computations reliably by 't', the throughput is thus quantified as

$$R = \frac{N}{[3N + 2(k-1) + p(k+2)] \cdot t} \tag{7.1}$$

This expression is also used to denote the throughput for non-MAC operations executed in the highly pipelined architecture.

B. MAC with unit delay in the Feedback Path

We next analyze the throughput for interval MAC operations performed in the highly-pipelined design using scheme 2 described in Section 5.2.2 since it incorporates

a delay in its MAC feedback path. Along the lines of the analysis for normal MAC operations in pipelined designs, the scenario of overflow and the corrective action is indicated in Figure 7.4.

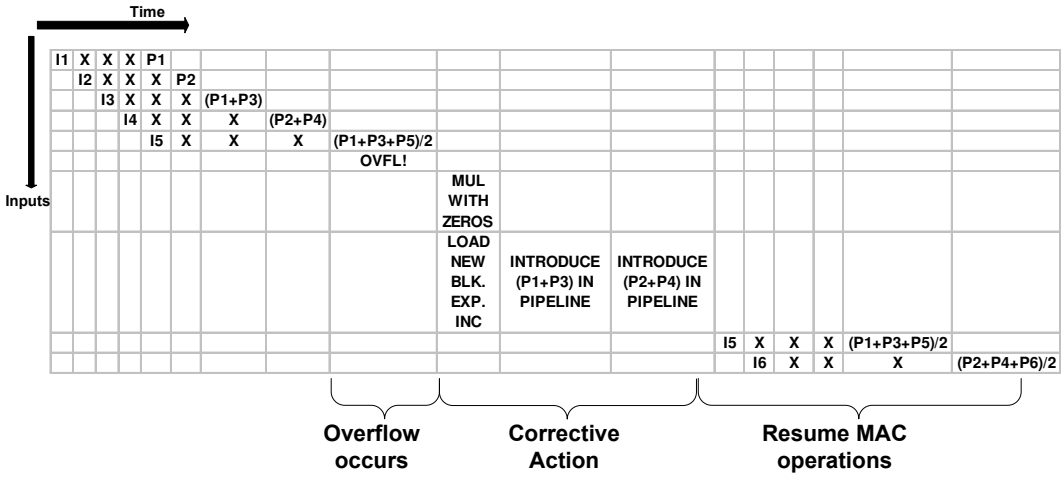


Figure 7.4: Post-overflow Corrective Actions : MAC pipelines with single delay

For a data block of size 'N', a pipeline design of 'k' stages and 'p' overflows, we still require (N+k-1) cycles for Exponent Detection and (N+k-1) cycles for left shifting. We may apply the inputs for MAC computations without waiting for the last sample of the data block to be left shifted. This is accounted for by the subtracted term (k-1) in the expression that denotes the time to process the samples. Each overflow requires corrective action as mentioned in Section 5.2.2. Clearing the pipeline by multiplication with 0s, loading the new output block exponent value, introducing each accumulated term in the pipeline - all take one clock cycle each. The penalty for every overflow that occurs is found to be (k+3) clock cycles - one more in than that of MAC pipelines that do not incorporate delays in the feedback path. This is attributed to the need for loading two terms in the MAC pipeline as opposed to one term previously. Furthermore, since the partially accumulated sums circulate in the feedback path alternately, they must be summed up in the end to yield the final result. This requires one more clock cycle. Therefore, the total number of clock cycles it takes to compute the output for the highly pipelined design is given by {(N+k-

1)+[(2N+k-1)+p(k+3)+1]} or [3N+2(k-1)+p(k+3)+1]. It must be noted that the highly-pipelined design has k=4.

The throughput for MAC operations in the highly-pipelined design is quantified as

$$R = \frac{N}{[3N + p(k+3) + 2(k-1) + 1] \cdot t} \tag{7.2}$$

We can assume that when N >> k, the penalty in computation time due to the overflows can be ignored. Therefore, the limit of throughput as N $\rightarrow \infty$ for both Equation 7.1 and 7.2 is found to be

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

We can plug the values for t from the timing results and obtain the best throughput of 86.1M samples/second for the highly pipelined design since its timing of 3.87ns is the least among the designs considered.

7.1.4 Type-2 : Non-MAC operations in Pipelined Designs

Throughput for Type-2 operations in a k-stage pipelined design is evaluated based on the assumption that Type-2 operations do not run into overflow. We devote (N+k-1) clock cycles for Exponent Detection, (N+k-1) clock cycles for Left shifting and [(N+k-1)-(k-1)] clock cycles for the computations. Therefore, the throughput is denoted by

$$R = \frac{N}{[3N + 2(k-1)] \cdot t} \tag{7.3}$$

We can assume N>>k and take limits as N $\rightarrow \infty$ to obtain the throughput as 1/(3t). This will also apply to addition and subtraction when overflow does not occur. This is true for both pipelined designs formed using pipelined multipliers as well as the highly pipelined design. The highly-pipelined design records the highest throughput of 86.1M samples/second among all designs considered in the design space. We observe that the expression for throughput for non-MAC operations is the same as that obtained for the MAC operations.

Figure 7.5 shows a comparison of throughput across all the pipelined designs. The black line joins the throughput points for designs obtained using pipelined multipliers only, for breaking the critical path. The highest point in the plot corresponds to the throughput of the highly-pipelined design which is superimposed on this plot. This corresponds to a throughput of 86.1M samples/second. This is an improvement of 166.9% over the throughput of the non-pipelined design of 32.2M samples/second.

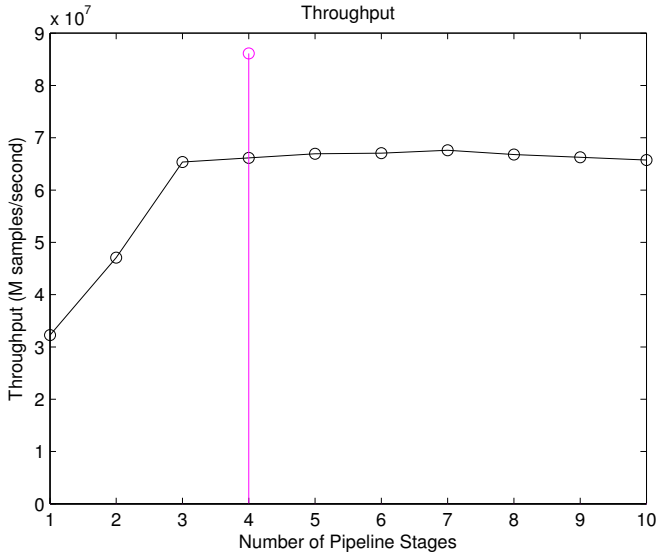


Figure 7.5: Throughput Across Pipelined Designs

Choice of Best Design

The best design in the available design space, comprised of different superpipelined architectures, is identified on the basis of the maximum throughput value for the intended application. The highly-pipelined design, with four pipeline stages, turns out to be the design with the maximum throughput. Therefore, it is chosen as the best design for signal processing applications.

7.2 Structural Hazards

This section describes dependencies associated with this architecture while performing *point-wise* computations that result from the sharing of computational resources. This section is of significance because it can affect the throughput of the system directly. This hazard does not apply to interval computations. This dependency exists irrespective of the stage of pipelining of the architecture. By definition, a *structural dependency* occurs if two or more requestors want to share the same resource at the same time, also known as a *collision*. Structural dependency is observed in this architecture when the ALU is in the *point̄wise* mode of operation. Both the Lower and Upper Bound modules share the same input lines X_L, X_U, Y_L and Y_U, and each command uses a specific set of inputs. Hence, this results in a input-line resource dependency between the modules. This is illustrated in the following example. In the *point̄wise* mode, if the Lower Bound is performing subtraction (X_L-Y_U), independent operations can only be performed in the Upper Bound module that do not involve X_L and Y_U. Hence operations such as *width* and *addition* cannot be performed in the Upper Bound module in parallel with subtraction in the Lower Bound module if the intention is to perform independent operations on different data. Table 7.1 shows the inputs that are involved with each command in the Lower and Upper Bound modules. The commands are shown as functions of specific inputs that they act upon. The operation of *multiplication* has not been mentioned deliberately because it has nine cases which are decoded at the time of execution. If we know *a priori* as to which

case of multiplication will be performed, then the information regarding dependencies may be extracted from Table 2.1.

Table 7.1: Commands Function Table (Point-wise Operations)

Command	*Lower Bound module*	*Upper Bound module*
ADD	$f(X_L,\ Y_L)$	$f(X_U,\ Y_U)$
SUB	$f(X_L,\ Y_U)$	$f(X_U,\ Y_L)$
DIV	$f(X_L)$	$f(Y_L)$
UNION	$f(X_L,\ Y_L)$	$f(X_U,\ Y_U)$
INTERSECTION	$f(X_L,\ Y_L)$	$f(X_U,\ Y_U)$
WIDTH	$f(X_L,\ X_U)$	$f(Y_L,\ Y_U)$
MIDPOINT	$f(X_L,\ X_U)$	$f(Y_L,\ Y_U)$
MIN	$f(X_L,\ Y_L)$	$f(X_U,\ Y_U)$
MAX	$f(X_L,\ Y_L)$	$f(X_U,\ Y_U)$
OR	$f(X_L,\ X_U)$	$f(Y_L,\ Y_U)$
AND	$f(X_L,\ X_U)$	$f(Y_L,\ Y_U)$
XOR	$f(X_L,\ X_U)$	$f(Y_L,\ Y_U)$
EXP. DET.	$f(X_L)$	$f(Y_L)$
NORMALIZATION	$f(X_L)$	$f(Y_L)$
LEFT SHIFT	$f(X_L)$	$f(Y_L)$

7.2.1 Dual MAC structure for pointwise computations

While structural hazards can lower throughput if we wish to perform independent operations in parallel, they can be useful and actually provide speedup for applica-

tions that can share input lines. Applications like filtering in signal processing utilize dual MAC structures to attain higher throughput when compared to the single-MAC structures. An estimate of the benefit from this structure may be obtained by analyzing the speedup for common applications such as filtering. This section illustrates that structural dependency is not always undesirable and estimates the speedup from sharing terms between dual parallel MACs in order to realize an FIR filter.

Sharing terms between the MACs

This section presents the evaluation of the speedup gained by sharing terms between parallel computations in a dual MAC structure. For an FIR filter with coefficients h and input x, the difference equation is given by

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

This equation may expanded as shown in Figure 7.6.

y(n)	=	*h(0)*x(n)*	+	*h(1)*x(n-1)*	+ ... +	*h(N-1)*x(n-N-1)*
y(n-1)	=	*h(0)*x(n-1)*	+	*h(1)*x(n-2)*	+ ... +	*h(N-1)*x(n-N-2)*
y(n-2)	=	*h(0)*x(n-2)*	+	*h(1)*x(n-3)*	+ ... +	*h(N-1)*x(n-N-3)*
y(n-3)	=	*h(0)*x(n-3)*	+	*h(1)*x(n-4)*	+ ... +	*h(N-1)*x(n-N-4)*

Figure 7.6: Difference Equation for an FIR filter

In a dual MAC structure, terms may be shared so that multiple computations can be performed in parallel. In DSPs, this scheme is known as in *time-based loop unrolling* [37]. The existence of two parallel MAC units in the Lower and Upper Bound modules allows us to compute *y(n)* and *y(n-1)* at the same time with shared coefficients when the ALU is in the *pointwise* mode of operation. However, structural dependency

must be kept in mind while scheduling operands to this ALU. For example, using the *a priori* knowledge that the coefficient *h(0)* and input samples *x(n)* and *x(n-1)* are positive, we may directly apply the inputs as shown in Figure 7.7. Such sharing of

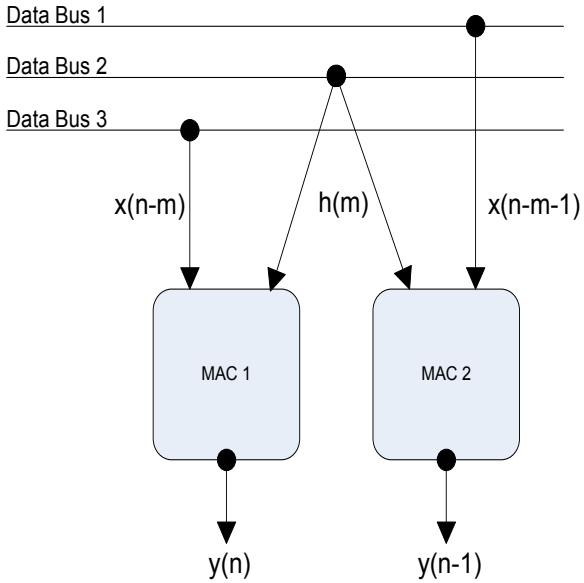


Figure 7.7: Feeding inputs to the Dual MAC structure for an FIR Filter

coefficient terms leads to twice the performance that could be obtained from a single MAC system in the best case. This is shown analytically in the ensuing lines.

Consider, for example, the system of difference equations indicated in Figure 7.6. For an FIR filter with N coefficients, an ALU with a single MAC structure would require N multiply-accumulate operations to compute *y(n)* and another N multiply-accumulate operations to compute *y(n-1)*. Hence, on the basis of the assumption that each computation requires only one clock cycle, the total computation time for this is taken to be 2N.

With the dual MAC structure, we can compute the grouped terms indicated in Figure 7.6 in each clock cycle. Hence, by the end of N clock cycles only, both *y(n)* and *y(n-1)* are available. The speedup in this case is computed as

$$\text{Speedup} = \frac{2N}{N} = 2$$

While the speedup factor of 2 corresponds to the case of *no* structural dependency, situations may arise that require the structural dependency to be resolved. Delaying the release of a command to the ALU until the dependency is no more applicable is one solution. This however reduces the speedup obtained. This work does not investigate the resolution of structural hazards due to this structural dependency.

In the ideal case where structural hazards do not occur, the throughput of the BFPIALU is double the value estimated in Section 7.1.2 because the point-wise operations are performed in parallel. Therefore, MAC operations record best throughput of 25.8x2 = 51.6M samples/second. Type-2 point-wise operations record a throughput of 86.1 x 2 = 172.2M samples/second. The process of arriving at an exact value of throughput for an application depends upon the distribution of commands in the computer program to be executed. This completes the discussion of the structural dependency associated with implementing point-wise operations in the BFPIALU.

7.3 Error Analysis

This section presents the investigation of the improvement in accuracy by using the technique of CBFS as opposed to using Input scaling for point-wise computations. Input Scaling involves scaling the samples in a data block for every Q0.15 fixed point addition or subtraction, so that the result of these operations on the scaled data will never result in an overflow. In contrast, CBFS performs the operation and then checks for overflow. The result is scaled down only if there is overflow. We describe the experiment that quantifies and compares the accuracy in both these methods of overflow handling for a simple 2^{nd} order filter.

Second order filter sections are important because they can be cascaded to yield

higher order filters with better performance in terms of lower round off errors. Intuitively, we expect that CBFS should provide better accuracy than Input scaling because the former performs conditional scaling and hence the loss of bits is subject to the occurrence of overflow.

7.3.1 The Input Data Sequence

An input sequence of data quantized to 16-bits in the Q0.15 fixed point format is taken into consideration to evaluate the accuracy of BFP computations with both Input Scaling and CBFS. Figure 7.8 illustrates this input sequence. This data was chosen due to the following characteristics:

- The samples bear values about both sides of 0 symmetrically. Hence both positive and negative valued data are taken into account.
- The samples range in values that cover the full dynamic range available in Q0.15 fixed point format.

Division into Data Blocks

Before applying the input samples, data is divided into blocks. The number of blocks in this case is chosen arbitrarily to be 2 based on an arbitrarily chosen block size of N = 579. Data in Block 1 occupies the full dynamic range available in Q0.15 format. However, data in the second block has different amplitude ranges. Data is distributed in this block between ranges of $[2^{-4}, -2^{-4}]$, $[2^{-8}, -2^{-8}]$ and $[2^{-12}, -2^{-12}]$.

7.3.2 Input Scaling Error Analysis

The difference equation for a 2^{nd} order FIR filter is given by :

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2)$$

Evidently, each output sample will require two additions to be performed. Therefore,

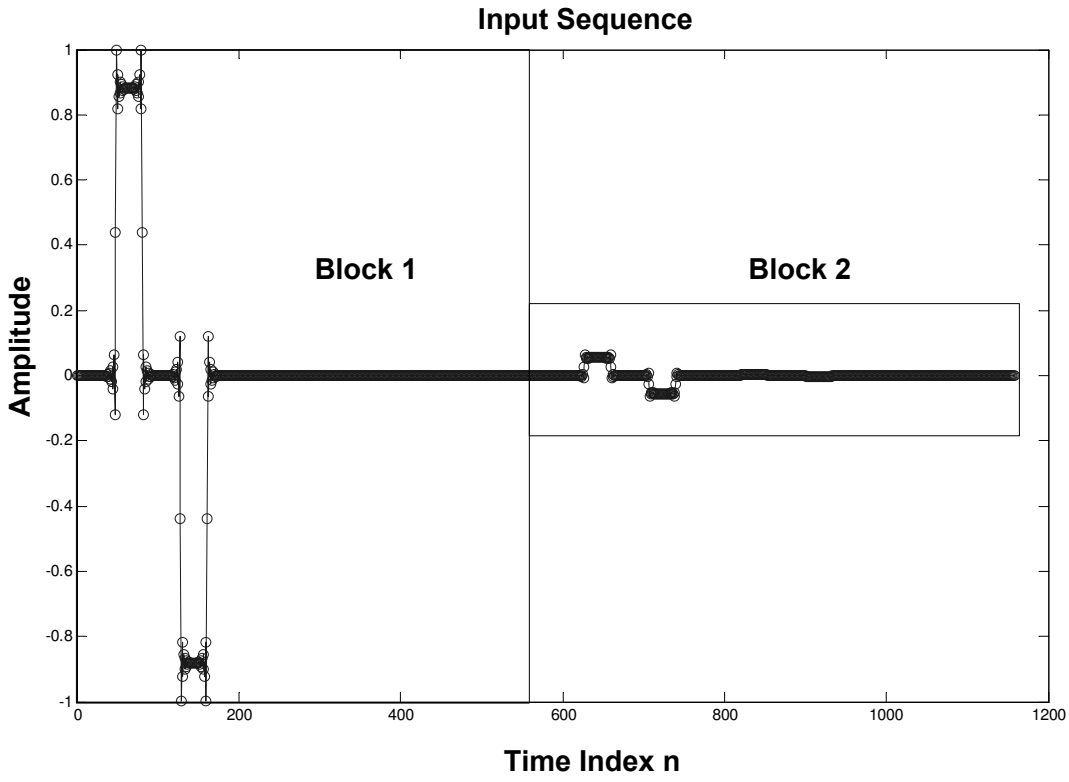


Figure 7.8: Input sequence for Error analysis

the unconditional scaling factor S is chosen to be equal to 2 in anticipation of a bit growth of 2 bits on the MSB side. It is applied to both blocks during normalization.

The filtered outputs are computed using direct fixed point additions of the three product terms to realize the difference equations. The filter coefficients *h(0), h(1)* and *h(2)* for the difference equation have been generated using the *fir1* routine in MATLAB. The filtered outputs of Block 1 and Block 2 are aligned and added to obtain the final output values. To calculate the error, the input and the coefficient values are quantized to the Q0.15 fixed point format first. Then, the convolution is performed treating these quantized values as *double* values. The simulation is performed with the 'floor' rounding mode in MATLAB. This gives the output expected on a Q0.15 hardware implementation with both the Lower Bound module and Upper Bound module in truncation mode. The experiment is repeated for different values

of cutoff frequencies ω in order to verify that the results are stable. The error due to Input Scaling technique is captured through the Mean Squared Error (MSE) shown in Table 7.2. The MSE for 'L' output samples is evaluated as

$$\text{MSE} = \frac{\sum_{i=1}^{L} (Output_{obtained} - Output_{ideal})^2}{L}$$

Table 7.2: Error with Input Scaling

ω	$Error - Input\ Scaling$ (10^{-9})
0.1	6.068170216192682
0.2	6.206350510524984
0.3	5.955736161763485
0.4	5.784091325519226
0.5	5.771197428900149
0.6	5.669805529570784
0.7	5.916439896294861
0.8	6.845908393080973
0.9	6.561484764213056

7.3.3 CBFS Error Analysis

The CBFS scheme is evaluated for error in this section. Both the input data blocks are normalized using S=0. Each output sample requires two additions in the realization of its difference equation. The occurrence of overflow is checked for after each operation of addition or subtraction. If overflow is detected, then a scaling is performed on the output side and the next input is scaled down by half. The final

filtered output is obtained by aligning and adding the two filtered outputs.

Under the CBFS scheme, we intuitively, expect better accuracy due to *conditional* scaling. In order to measure the error in this case, the comparison is performed against filtered output obtained in *double* format with the input and coefficients quantized to Q0.15 format. Again, the 'floor' rounding mode is used in the MATLAB simulation assuming that both the Lower Bound and Upper Bound modules are in the truncation mode. The MSE is quantified for different values of the cut off frequency ω as shown in Table 7.3. The MSE is quantified for 'L' output samples as

$$\text{MSE} = \frac{\sum_{i=1}^{L} (Output_{obtained} - Output_{ideal})^2}{L}$$

Table 7.3: Error with CBFS

ω	$Error - CBFS$ (10^{-9})
0.1	0.3553081281486225
0.2	0.3541738510678708
0.3	0.3476477889902497
0.4	0.3322888835019018
0.5	0.3497997436420356
0.6	0.3737267276109385
0.7	0.3689648541008730
0.8	0.3745123327633928
0.9	0.3742518436654803

7.3.4 Comparing Input Scaling and CBFS

Figure 7.9 illustrates the overlapping filtered outputs in the ideal case, Input Scaling and the CBFS technique for $\omega = 0.5$. The MATLAB code used to perform this simulation is presented in the Appendix.

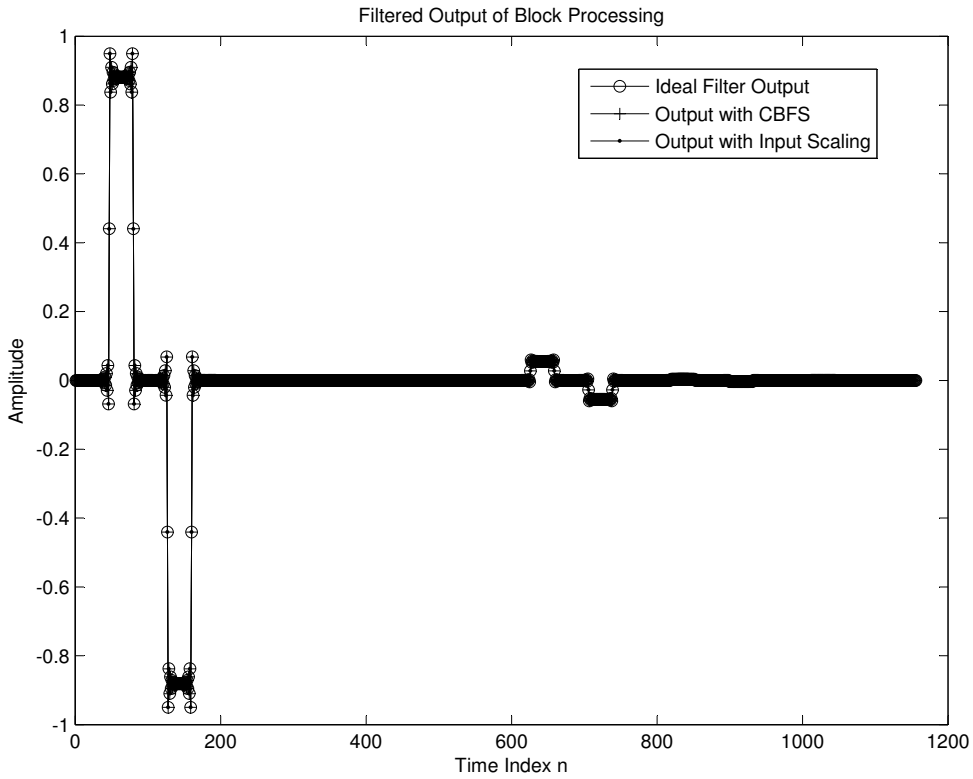


Figure 7.9: Filtered output : Input scaling vs CBFS

The results from the experiments in the case of Input Scaling and CBFS is presented collectively in Table 7.4. Figure 7.10 illustrates the comparison of error for the Input scaling and the CBFS techniques. The average factor of improvement in accuracy is 17 when the technique of CBFS is used in place of Input Scaling. This clearly justifies the use of CBFS in this architecture for pointwise operations since this implies a better signal to noise ratio and the BFPIALU is intended to be used in signal processing and control applications. The MATLAB code for the same is

presented in the Appendix. Routines for Exponent Detection by XORing successive bits of the input samples are not presented since it is trivial to code it up.

Table 7.4: Error Comparison of Input Scaling and CBFS schemes

ω	$Error$:$Input\ Scaling(10^{-9})$	$Error$:$CBFS(10^{-9})$	$Improvement\ Factor$
0.1	6.068170216192682	0.3553081281486225	17.07861356
0.2	6.206350510524984	0.3541738510678708	17.52345774
0.3	5.955736161763485	0.3476477889902497	17.1315232
0.4	5.784091325519226	0.3322888835019018	17.40681561
0.5	5.771197428900149	0.3497997436420356	16.49857535
0.6	5.669805529570784	0.3737267276109385	15.1709929
0.7	5.916439896294861	0.3689648541008730	16.03523975
0.8	6.845908393080973	0.3745123327633928	18.27952725
0.9	6.561484764213056	0.3742518436654803	17.53227105

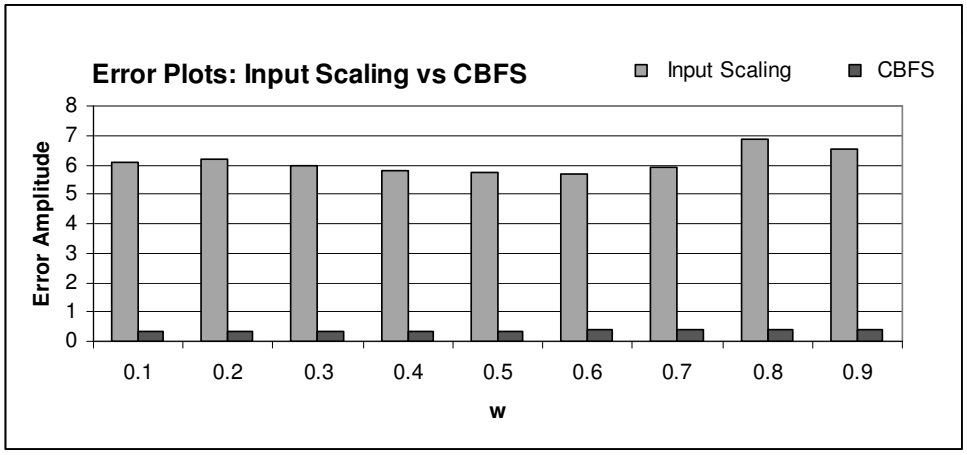


Figure 7.10: Error Plots for CBFS and Input Scaling

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis presents a 16-bit Block Floating Point Interval ALU (BFPIALU) architecture for DSP and control applications to address the issue of small dynamic range in dedicated two's complement fixed point implementations of the same. The BFP scheme is proposed in order to avoid wrap around in dedicated fixed point architectures which can lead to unreliable bounds with interval arithmetic.

The design decisions for the proposed BFPIALU architecture are taken on the basis of a set of criteria, namely *correctness, closedness, totality, optimality, efficiency* that assure the reliability of interval operations. The BFP arithmetic scheme is adopted as a means to achieve a dynamic range on the fixed point architecture higher than the conventional fixed point would allow. BFP arithmetic is a memory based scheme and all data samples are read in from a memory (such as a Dual Port RAM for system memory or even high speed local RAMs). BFP support in the BFPIALU is provided through special commands such as Exponent Detection and Normalization so that blocks of data can be normalized. Recognizing that overflow

has to be handled properly, this work compares the techniques of Input scaling and Conditional Block Floating point Scaling (CBFS) and chooses CBFS for implementation because it provides better accuracy and provides optimal interval bounds. The experiment performed to determine the same involves a 2^{nd} order FIR filter section. Simulations are run on the filter for different sets of filter coefficients to collect statistical results of accuracy. Overflow detection is performed for operations of addition, subtraction and multiply-accumulate because they lead to bit growth on the MSB side in the interval ALU as part of CBFS.

This work expands the command set mentioned in [9] to include the comparison operations of MAX, MIN and logical commands of AND, OR and XOR. The ability to perform logical operations is important for applications such as error control coding and sensor control. Comparison operations find application in interval-based fuzzy systems. Modifications to the architecture presented in [9] include a reduction in the execution time for all operations in one clock cycle. The affected operations include interval multiplication, interval union and interval intersection.

The proposed design can also function as two parallel ALU structures performing point-wise computations. Logic is inserted to control the instructions dispatched to the Lower and Upper Bound modules based on the mode of operation of the ALU (*interval* or *point̄wise* mode). Thus, the interval architecture can be used for both interval operations as well as for point-wise operations. The outward rounding scheme is retained for interval operations. The point-wise mode of operation, however, extends rounding to $+\infty$ to the Lower Bound module and truncation to the Upper Bound module. Owing to the fact that *saturation arithmetic* is not feasible for interval arithmetic and that this work incorporates modifications from the existing rounding schemes, the saturation scheme has not been considered for the point-wise operations. Nevertheless, it forms part of the future developmental work in this area.

The non-pipelined architecture is synthesized and the least timing is found to be as low as 10.33ns or about 96.8 Million Interval Operations per Second (MIOPS).

This is extremely poor from the perspective of signal processing applications that demand high throughput. Hence, the design is subjected to pipelining to increase the throughput. Synthesis of the pipelined designs shows an increase in the area occupied by the designs with higher levels of pipelining. This is to be expected because more logic is inserted with the addition of every pipelined stage. The minimum clock period is seen to fall till the seventh stage of pipelining at 4.93ns. Due to the law of diminishing returns, the clock period then begins to increase after this stage. Then, registers are inserted manually and the timing improvements are observed. The highly pipelined architecture can perform reliably at a clock period of 3.87ns. This corresponds to 258.4 MIOPS. Timing improves by 166.9% from the non-pipelined design to the highly-pipelined design.

Throughput, however, cannot be quantified as *operations per second* because it is a function of the number of pipeline stages, the number of overflows and clock rate. A Monte-Carlo simulation of random product accumulation shows that with each overflow, the probability of the next overflow decreases. Throughput is evaluated for a large block size across different pipelined architectures. The highly-pipelined design delivers the highest throughput of 86.1 M samples/second for interval operations whereas the non-pipelined architecture records a throughput of 32.2M samples/second. Evidently, throughput improves by 166.9% for the highly-pipelined design as compared to the non-pipelined design.

Given the ability of the BFPIALU to function as two individual ALUs that can perform point-wise operations, this work identifies a *structural dependency* which occurs due to the sharing of input lines between the Lower and Upper Bound modules of the interval ALU. This is estimated to affect throughput adversely for point-wise operations. It identifies a dual MAC structure within the architecture and estimates the associated speedup in example applications due to it in order to turn it to advantage. A simple FIR filter is evaluated for speedup. The best speedup is obtained when no structural dependency exists between the operations performed in the Lower and Upper Bound modules or when sharing terms between operations is feasible. The speed

of operation in this case is identified to be twice that of interval operations, equal to 258.4x2 = 516.8 million point-wise operations per second. This is the peak value and the deterioration in speedup is directly proportional to the amount of structural dependency in the operations. The throughput is found to be 172.2 Msamples/second for pointwise operations.

8.2 Future Work

This section lists the future work in this area by dividing them into short term and long term goals.

8.2.1 Short Term Goals

1. Since structural dependency can affect throughput adversely, devising useful methods to address this issue could be very useful. Architectural changes may have to be employed in the interval ALU to resolve this dependency.

2. Adding the Saturation scheme when overflow is detected for point-wise operations only is also one of the goals.

8.2.2 Long Term Goals

Modern DSPs rely on pipelining and parallelism to attain higher throughputs. The interval ALU developed in this work exhibits both these features. Hence, it could be very useful in the execution units of superscalar or VLIW Multi- ALU Processors. Development of the Instruction Set Architecture (ISA) for such a processor, the associated processor design and the construction of a compiler can help applications based on interval arithmetic immensely.

Bibliography

[1] E. Hansen and G. W. Walster, Global optimization using interval analysis. Marcel Dekker, Inc. and Sun Microsystems, Inc., 2004.

[2] Q. Liang and J. M. Mendel, "Overcoming time-varying co-channel interference using type-2 fuzzy adaptive filters," IEEE Transactions on Circuits and Systems - II, vol. 47, Dec 2000.

[3] W. D. Collins and C. Hu, "Fuzzily determined matrix games." Berkeley Initiative for Soft Computing 2005, 2005. University of Central Arkansas.

[4] R. B. Kearfott and V. Kreinovich, eds., Applications of Interval Computations. Netherlands: Kluwer Academic Publishers, 1996.

[5] R. Shettar, R. M. Banakar, and P. S. V. Nataraj, "Design and implementation of interval arithmetic algorithms," in Industrial and Information Systems, First International Conference on, (Peradeniya), pp. 328–331, Aug. 2006.

[6] http://www.cs.utep.edu/interval-comp/intsoft.html.

[7] U. Kulisch, Advanced Arithmetic for the Digital Computer. New York: Springer-Verlag, 2002.

[8] M. J. Schultz and E. E. Swartzlander, "A family of variable-precision interval arithmetic processors," IEEE Transactions on Computers, vol. 49, May 2000.

[9] R. Gupte, W. Edmonson, Jaya, S. Ocloo, and W. Alexander, "Pipelined alu for signal processing to implement interval arithmetic," Signal Processing Systems Design and Implementation IEEE, pp. 95–100, 2006.

[10] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, Synthesis and Optimization of DSP algorithms. Kluwer Academic Publishers, 2004.

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

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

[13] A. Akkas, "A combined interval and floating-point comparator/selector," Application-Specific Systems, Architectures and Processors, pp. 208 – 217, July 2002.

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

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

[16] A. Chhabra and R. Iyer, "A block floating point implementation on the tms320c54x dsp," tech. rep., Texas Instruments, December 1999. Application report SPRA610.

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

[18] 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, October 2002.

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

[20] A. C. Erickson and B. S. Fagin, "Calculating the fht in hardware," IEEE Transactions on Signal Processing, vol. 40, June 1992.

[21] www.actel.com/ipdocs/CoreFFT_DS.pdf.

[22] G. W. Gerrity, "Computer representation of real numbers.," IEEE Trans. Computers, vol. 31, no. 8, pp. 709–714, 1982.

[23] S. Ocloo and W. W. Edmonson, "An interval-based algorithm for adaptive iir filters," Fortieth Asilomar Conference on Signals, Systems and Computers, ACSSC, vol. 40, pp. 258–262, Oct 2006.

[24] T. Howe, "Implementing 3g layer 1 signal processing (tx)," tech. rep., Plextek Ltd. London Road, Great Chesterford, Essex, CB10 1NY.

[25] D. A. S. C. of the IEEE Computer Society, "Ieee std 1666 - 2005 ieee standard systemc language reference manual," pp. 0– 423, 2006.

[26] T. Aamodt, "Floating-point to fixed-point compilation and embedded architectural support," Master's thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, Canada, 2001.

[27] A. S. University, "Fixed-point arithmetic - part-ii." Real time Digital Signal Processing : EEE 404/591 Lecture Notes, 2007.

[28] S. M. Kuo and W.-S. Gan, Digital Signal Processors : Architectures, Implementations and Applications. Pearson Education, Inc, Upper Saddle River, New Jersey 07458: Pearson Prentice Hall, 2005.

[29] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, Synthesis of Arithmetic Circuits. John Wiley & Sons, 2006.

[30] http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html.

[31] H. G. Cragon, Memory Systems and Pipelined Processors. Sudbury, Massachusetts: Jones and Barlett Publishers, 1996.

[32] M. Vidal and D. Massicotte, "A vlsi parallel architecture of a piecewise linear neural network for nonlinear channel equalization," in Instrumentation and Measurement Technology Conference, 1999. IMTC/99. Proceedings of the 16th IEEE, vol. 3, (Venice), pp. 1629–1634, May 1999.

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

[34] Synopsys Design Compiler, Synopsys Corporation, http://www.synopsys.com/products/logic/design_compiler.html.

[35] http://www.synopsys.com/products/power/primepower_ds.pdf.

[36] http://www.ece.ncsu.edu/muse/sshaft/tut1.php.

[37] B. Varnagiryte, A. Zemelis, O.Olsen, P. Koch, O. Wolf, and E. Kazanavicius, "A practical approach to dsp code optimization using compiler/architecture," ULTRAGARSAS 2002, vol. 43, no. 2, 2002. Ultrasound Institute.