MSB-First Interval-Bounded Variable-Precision Real-Time Arithmetic Unit

Yusrila Y. Kerlooza[1], Yudi S. Gondokaryono[2] & Agus Mulyana[3]

[1,3]Computer Engineering Dept., UNIKOM
[2]School of Electrical Engineering and Informatics, Bandung Institute of Technology

Abstract. This paper presents a paradigm of real-time processing on the lowest level of computing systems: the arithmetic unit. The arithmetic unit based on this principle containing addition, subtraction, multiplication and division operations is described. The development of the computation model is based on the Soft Computing and the Imprecise Computation paradigms, combined with the MSB-First and the Interval Arithmetic techniques. Those paradigms and techniques give the arithmetic unit design the ability to compute with precisions as a function of time available or accuracy needed. The predictability of processing time and result's accuracy are obtained by means of processing granularity of k-bits and by using look-up tables. We present an evaluation of the operation in time delay and computation accuracy that shows significant performance improvement over conventional arithmetic unit architecture, that is, the ability to produce intermediate-result during execution time, to give certainty in computation accuracy even before the process finish time by providing two intermediate-results, which act as the lower and upper bound of the real and complete computation result, and finally, gain high computation accuracy from the early time of the execution process.

Keywords: *arithmetic unit; interval-bounded; MSB-first; real-time; variable-precision.*

1 Introduction

1.1 The Real-Time Systems

A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness [1]. From the point of view of the real-time computation in the hardware level, most present-day strategies are focused on increasing hardware computational performance by using parallelism, segmentation or multiprocessing design techniques in order to decrease the average response delay.

These strategies are not always the most suitable ones for solving certain problems and they give rise to a multitude of questions: in the demand for requirements of reduced size applications, is the incorporation of multiprocessor architectures embedded in the system acceptable? For minimum timing constraint applications, can a logically correct decision be made only on an imprecise numeric result? Does adaptation to changes in environmental requirements require the system architecture to be redesigned? The investigation described in this paper considers these questions in the current implementations of calculation techniques and proposes a real-time architecture for arithmetic calculations that adapts the processing delay to the required time of the task.

1.2 Soft Computing and Imprecise Computation

Zadeh [2] claims in his paper on soft computing, that the real world is pervasively imprecise and uncertain, and that precision and certainty carry a cost. This statement is relevant with the issue studied in this paper, i.e., how a real-time system can achieve accurate result in conditions that there are often not enough time to compute all the operand's precision. We have to design a system that exploits the imprecision and uncertainty in order to achieve robustness, tractability, and low solution cost.

Another idea to solve the issue comes from the imprecise computation model studied by numerous researches [3-7]. It is a flexible technique for the design of real-time systems scheduler that are subject to overload. Each task is decomposed into a mandatory part followed by an optional one. The first part represents the minimum amount of processing necessary to obtain an acceptable result; the second one refines this result and reduces the rate of error.

The aim of this paper is to make progress in the incorporation of temporal restrictions in arithmetical basic operators; that is, to introduce real-time properties into the low level of arithmetic hardware, making use of the imprecise computation model and the predictability of response time and accuracy provided by access to look-up tables. This paper is focused on specific aspects of adjustable calculation of the arithmetic unit architecture and the operators it provides, which serves as a basis for designing other generic models of low level real-time schedulers. This paper is a continuation of previous researches on real-time arithmetic made by Mora, et al. [8], Kuspriyanto and Kerlooza [9-15].

2 Design Principles

Our design objective for the arithmetic unit is to build an architecture that includes features to support timing and accuracy constraints. The proposal consists of combining three techniques:

1. Computing from the most significant bit (MSB-First) and increasing the granularity of the elemental operator,
2. Producing two values that indicate lower and upper bounds of the actual numeric result, and
3. Obtaining the result in an incremental way.

2.1 MSB-First Computation and Greater Operator Granularity

Conventionally, computation process is carried out by a computer from the least significant bit first (LSB-First) just like we calculate, thus this technique gives slow numeric accuracy escalation throughout the process (see Figure 1).

Nielsen and Kornerup [16] conducted research on MSB-First digit serial arithmetic and our previous research on MSB-First arithmetic architecture [11-15] shows the potential advantage of this technique over conventional ones. Those previous researches also show the need of the intermediate-result: a successive product of ongoing arithmetic process execution that can be accessed by other computation tasks or elements during process time.

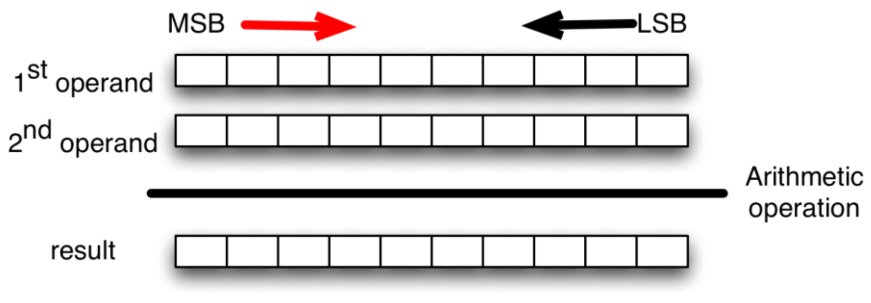


Figure 1 The calculation concept of LSB-First vs MSB-First.

Let's define the first and second operand as X and Y consecutively. Each operand consists of *n* bits and $\otimes$ denotes the arithmetic operations, $c_{t.ars}$ as the time constant to process single pair of operands' bit needed by the *ars* arithmetic unit, and t_a and $t_{f.ars}$ as the start and finish time of the arithmetic execution of the *ars* arithmetic unit, then we can find $m(t)$ the total bits that have been computed as a function of time:

$$m(t) = \left\lfloor \frac{t - t_a}{c_{t.ars}} \right\rfloor \quad \text{for } t_a \le t \le t_{f.ars} \tag{1}$$

We can also define the numeric value of the arithmetic computation carried out by the *ars* arithmetic unit $\mathrm{N}(t)_{ars}$ for the LSB-First and MSB-First computation as Equation (2) and Equation (3) consecutively.

$$N(t)_{LSB} = \sum_{i=0}^{m(t)-1} x_i \otimes y_i \tag{2}$$

$$N(t)_{MSB} = \sum_{i=n-m(t)}^{n-1} x_i \otimes y_i \tag{3}$$

Based on Equation (2) dan Equation (3), we can predict how the two technique performance in gaining numeric value in arithmetic operation as depicted in Figure 2. To maximize the advantage of the MSB-First computation, the incomplete result (we call it the intermediate-result) should be able to be accessed during the computation time.

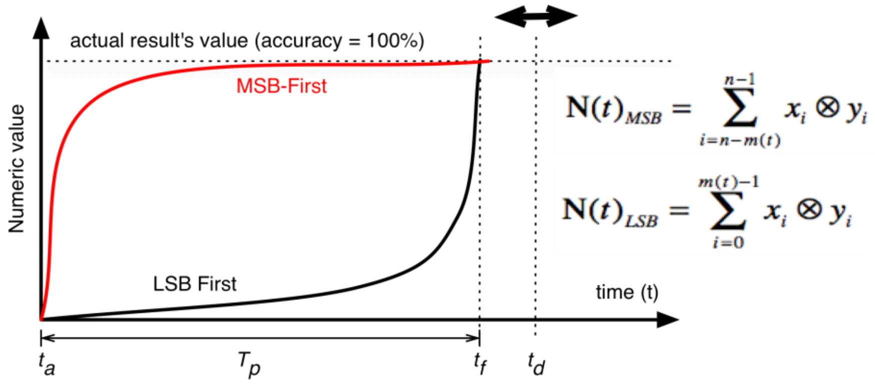


Figure 2 The performance of LSB-First and MSB-First computation.

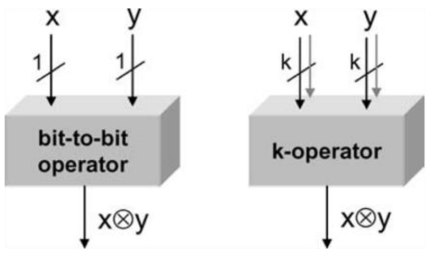
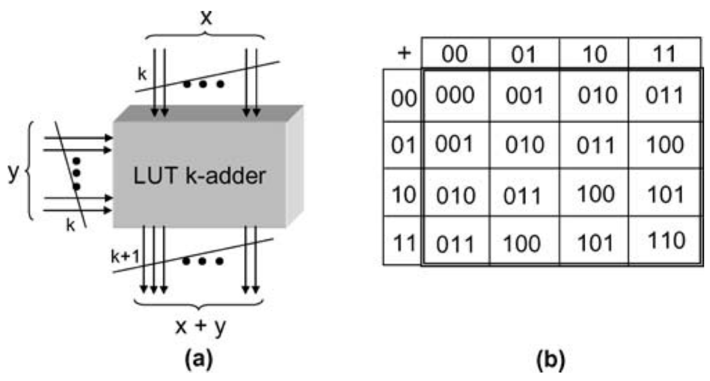


Figure 3 Bit-to-bit vs *k*-operator [2].

As Mora [8] said, most elementary operators consider a bit to be the minimum unit of information that can be processed. They are called bit-to-bit operators. A forward step consists of increasing the granularity and taking a group of bits as the minimum unit of operation. In this paper, we consider *k*-operators as the elementary operators that take a *k*-bit as the minimum unit of information that can be processed. Figure 3 schematically shows the functionality of a generic *k*-operator. The fundamental idea lies in obtaining advantages in the design of the generic arithmetic operators by using *k*-operator elements in their construction, which contribute to their adjustable processing. The *k*-operator designs may offer inherent improvements in bit-to-bit operations: the structure of the arithmetical unit is simplified when using fewer individual processing units to process groups of bits. The bit-to-bit operators are a particular example of *k*-operators with *k*=1.

The *k*-operators present several design alternatives, the most intuitive one consists of making a design based on combinational logic. The combinational circuit will produce the result of the function for *k*-size operands. Alternatively, we must make the most of electronic technology by searching for new proposals that would probably have been prohibitive some time ago, but not at present. We search for an implementation that provides a predictable operator response time. So, a general design technique for the *k*-operators resides in using look-up tables (LUT) to make the effective calculation. In this way, for any pair of blocks of *k* bits, the memory structure contains the direct result of its operation. These look-up tables must store all the results for *k*-size operands so that it is only necessary to select the cell that contains the result. The operand value itself is used to address the table. The nature of the stored data will depend on the function to be calculated. Figure 4 schematically shows a *k*-bit adder based on memory-oriented designs.



+	00	01	10	11
00	000	001	010	011
01	001	010	011	100
10	010	011	100	101
11	011	100	101	110

Figure 4 (a) Block diagram of a *k*-bit adder. (b) Table content for *k*-bit adder, with *k*=2 [2].

In this way the computation delay for each pair of blocks is similar, irrespective of its value, and, what is more important for our purposes, the time delay of the complete operation is a multiple of this time delay.

2.2 Interval Bounded

If we look again to Figure 2, both LSB-First and MSB-First techniques cannot tell us its computation accuracy before t_f. We can add the ability to predict where the final computation value lies by using the same idea of the interval arithmetic methodology introduced by Moore and Yang [17], Moore [18], and Boche [19]. The interval arithmetic produces two values for each arithmetic operations. The two values correspond to the lower and upper endpoints (bounds) of an interval, such that the true result is guaranteed to lie on this interval. The width of the interval, i.e., the distance between the two endpoints, indicates the accuracy of the result. Interval arithmetic was originally proposed as a tool for bounding rounding-off errors in numerical computation [18]. It is also used to determine the effects of approximation errors and errors that occur due to non exact inputs. Interval arithmetic is especially useful for scientific computations in which data ara uncertain or can take a range of values.

We can produce lower and upper bounds for LSB-First and MSB-First by adopting several algorithms. One of the simplest thing to compute the upper bound is by subtracting the maximum value of the arithmetic operation with the lower bound (computed by the original algorithm) in parallel. In this way during computation time, there will be two intermediate-results, which denote the lower and upper bounds of the true values. Figure 5 depicts the basic idea of the interval bounded concept.

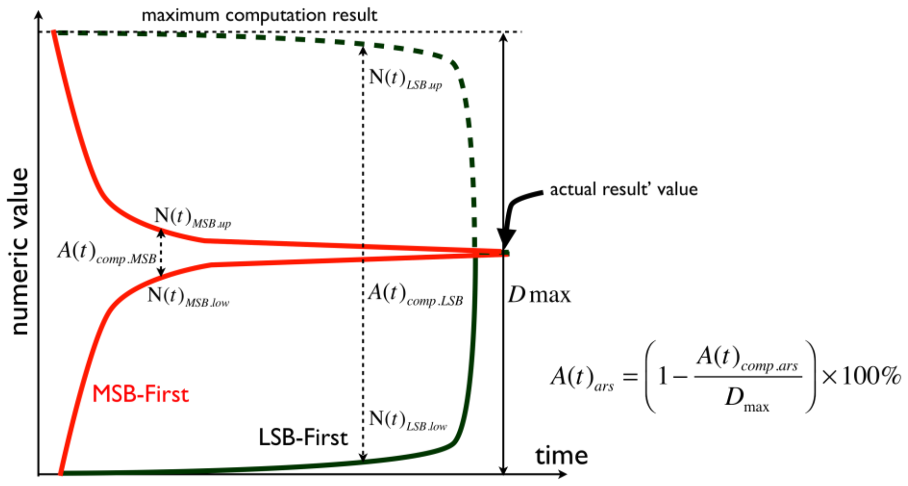
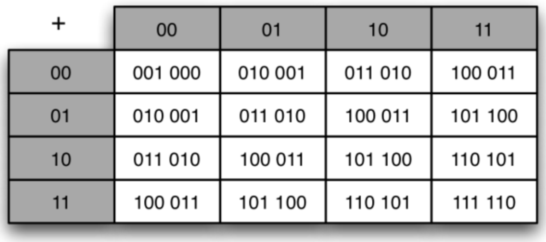


Figure 5 The interval bound concept of self-accuracy estimation by providing lower and upper bound value.

Using LUT-based computation we can actually make the difference between the lower and upper bounds (the accuracy) is predictable and no further computation needed to produce the upper bound value. The upper bound value in each *k*-operator can be stored on the same address with the lower bound value. Figure 6 shows the modification of the LUT content in Figure 4(b). In each address, the left three bits are the lower bound and the right three bits are the upper bound value of k=2 addition.



+	00	01	10	11
00	001 000	010 001	011 010	100 011
01	010 001	011 010	100 011	101 100
10	011 010	100 011	101 100	110 101
11	100 011	101 100	110 101	111 110

Figure 6 Modification of the table content for k-bit adder, with k=2.

2.3 Variable Precision

The delay adjustment ability and the variable quality of the result of each function depends on the possibility of partially executing its implementation. In general, each operator has a part that must be executed obligatorily and another that can be partially calculated [5,20]. The execution control of this optional part will allow us to adjust the function performance according to the application requirements (accuracy needed or available time). In this aspect, the implemented partial execution technique (stages or iterations) must provide capabilities for successive refinement of the solution and thus, support real-time requirements. Response delay is related to the number of calculated stages or iterations of the operations. Normally, a shorter process time results in less accuracy in the results. If the computation accuracy is met or the time left for computation is up, the execution can be stopped and intermediate-result can be accessed by other processes. This is the basic concept of variable precision computation covered in this paper.

3 Architecture

In this section, we develop an architecture of three basic arithmetic operations: addition/subtraction, multiplication and division in which timing constraints are present. Figure 7 shows the block diagram of the MSB-First Interval-Bounded Variable-Precision (MFIBVP) arithmetic unit.

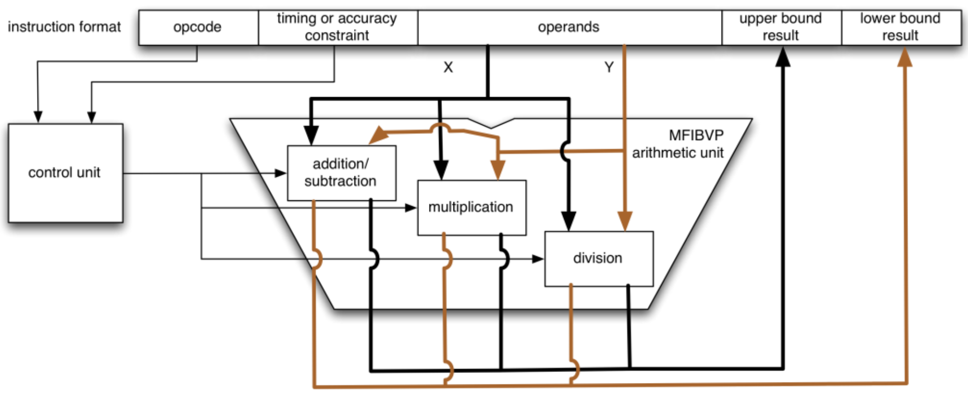


Figure 7 Block diagram of the MFIBVP arithmetic unit.

3.1 Addition and Subtraction

Similar to [8], the proposed addition method is based on the carry-select adder scheme and is made up of the following steps:

1. Fragmentation of operands into *k*-size blocks: It is immediate from the original operands. For operands with numbers of *n* bits (with $n > k$), we can divide the number into *n*/*k* blocks of *k* bits.
2. Addition of the corresponding pairs of blocks. The partial additions are obtained directly from a compound *k*-adder grid. This block contains a few *k*-adder operators that add the parts of the operands in parallel. Considering the reduced size of *k*, it is feasible to fit multiple *k*-adder operators into the grid block. The carry process is performed directly by obtaining the sum and its successor from the *k*-operator. An array of look-up tables is used in this paper as the *k*-adder operators.
3. Ordered concatenation of the partial additions taking the carry logics into account: The selection of each block is a function of the carry bit of the preceding block, selected according to the algorithm carry-select adder. For example, Figure 8 shows the operation scheme for operands fragmented into four parts. The resulting formation by means of successive selections of the partial sums can be observed.

The detailed schematic of the carry-select adder scheme for Figure 8 is depicted in Figure 9 (a-d). The MFIBVP adder can be converted into a subtractor with the same features by employing XORs with two fan-in in each bit of the second operand (assuming the first operand acts as the subtrahend). The first XORs input pins are connected to the second operand's bits and the others to the c_0. If the value of c_0=0 it will act as an adder, otherwise it will act as a subtractor, since the value of the second operand will be converted to its $2^{'s}$ complement.

Figure 10 shows the position of XORs in the adder's first step of its block diagram.

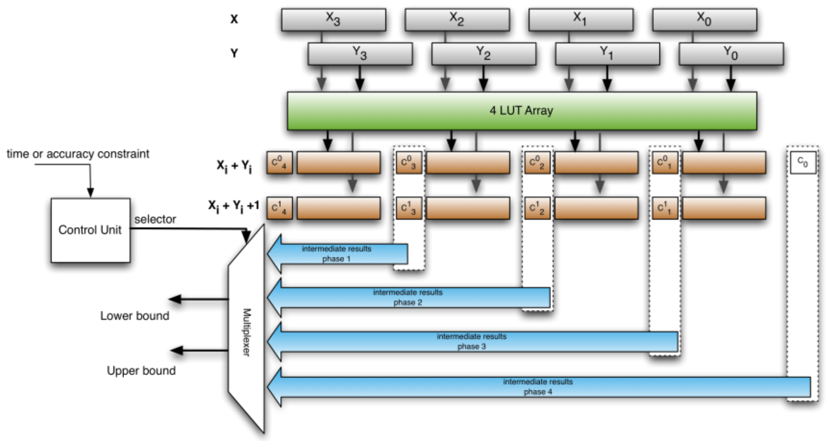

X
Y
X_3
X_2
X_1
X_0
Y_3
Y_2
Y_1
Y_0
4 LUT Array
time or accuracy constraint
$X_i + Y_i$
$X_i + Y_i + 1$
Control Unit
selector
intermediate results phase 1
intermediate results phase 2
intermediate results phase 3
intermediate results phase 4
Multiplexer
Lower bound
Upper bound

Figure 8 The basic concept of successive refinement of intermediate result of the MFIBVP addition.

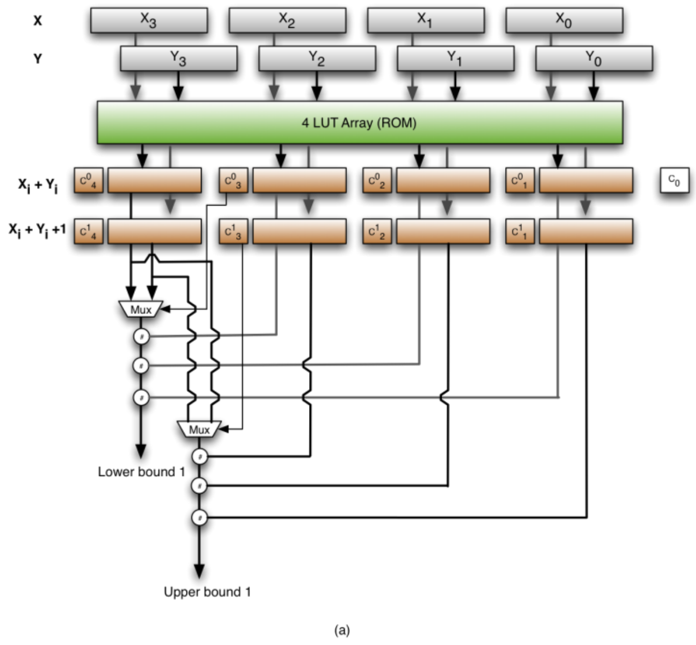
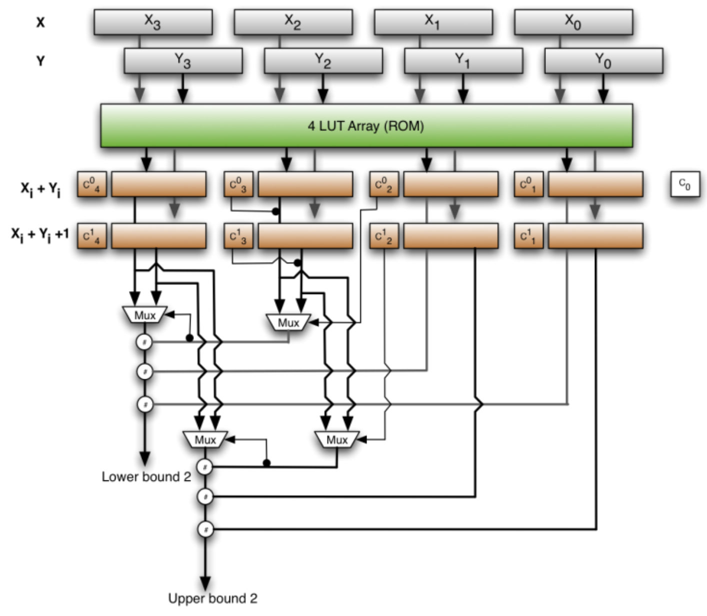
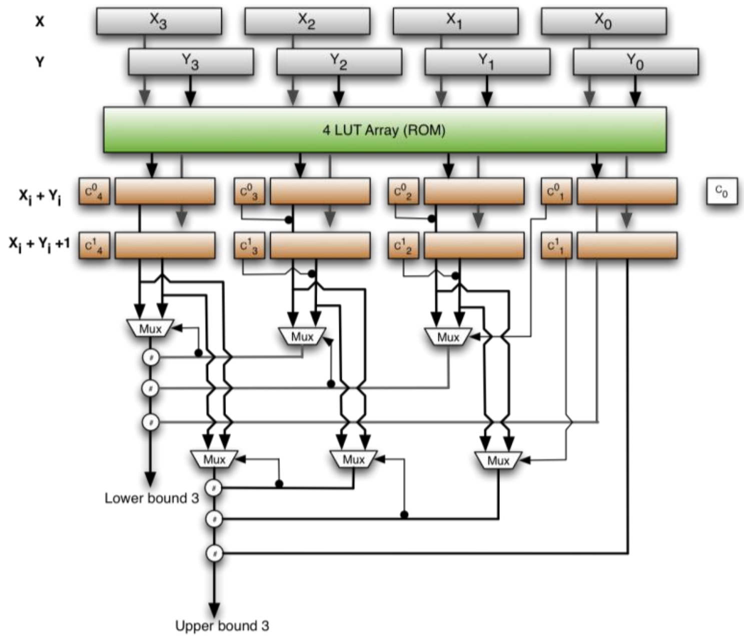


Figure 9 The MFIBVP adder schematic with n/*k*=4; (a to d) the fastest but least accurate to the slowest but most accurate computation ***(continued)***.



(b)



(c)

Figure 9 ***(continue)*** The MFIBVP adder schematic with n/*k*=4; (a to d) the fastest but least accurate to the slowest but most accurate computation.

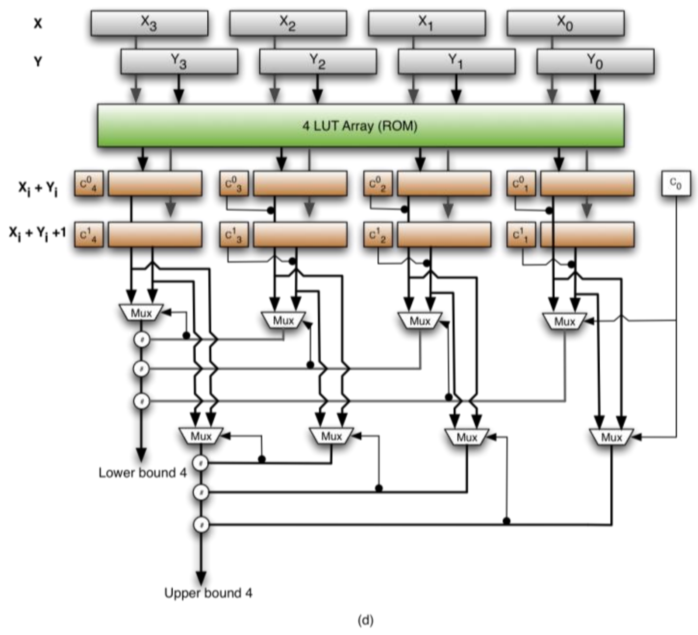


Figure 9 ***(continue)*** The MFIBVP adder schematic with n/*k*=4; (a to d) the fastest but least accurate to the slowest but most accurate computation.

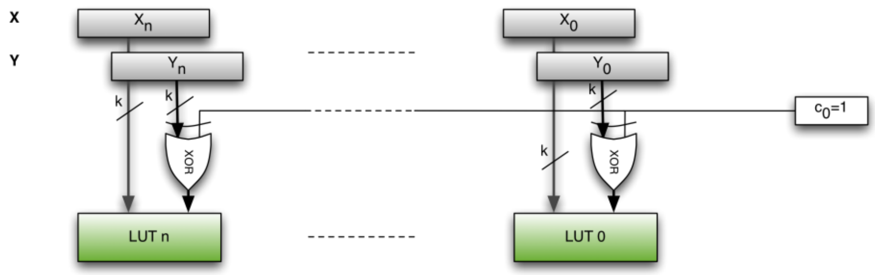


Figure 10 Additional XORs on the second operand will change the MFIBVP adder to a subtractor.

The MFIBVP adder design is based on the previous algorithm, with the special feature that only part of the blocks obtained from the operands are combined according to the time availability. As the Figure 8 depicts, the operation control line will select the appropriate result for each application. We propose that the sum and combination of the blocks will begin with the last block by considering the value of $c^0_{\frac{n}{k}-1}$ for determining the lower bound and the value of $c^1_{\frac{n}{k}-1}$ for determining the upper bound as shown in Figure 10(a), depending on timing constraints, and move towards the left. The rest of the blocks of the results are taken directly from the LUT array without being combined, causing no additional delay. Thus, depending on the application requirements, the system will adapt the quality-delay of the response. Of course, according to the increase in the number of selection stages, the error computation will decrease.

3.2 Multiplication

The first implementation of the multiplication operation depicted in Figure 11 is basically a well-known multiplier technique: the unsigned array multiplier [21]. It consists the following steps:

1. Generation of partial products: The partial products generation process is crucial to the operation's overall performance. Two aspects must be taken into account in its design: the complexity of the generating circuit and the number of partial products generated. The first aspect is linked to the time taken in generating each partial product, whereas the second one affects the time taken in the second step below to reduce them into two operands that will be added in the last step.
2. Reduction in the number of partial products: The general way in which a high performance multiplier works consists of combining the partial products in order to reduce their number until a total of two is reached. We can use Wallace tree method [22] for the reduction of the partial products.
3. Final addition: It can be implemented by well-known addition methods; nevertheless, due to the MFIBVP features, we have used the previously proposed adder, the MFIBVP adder.

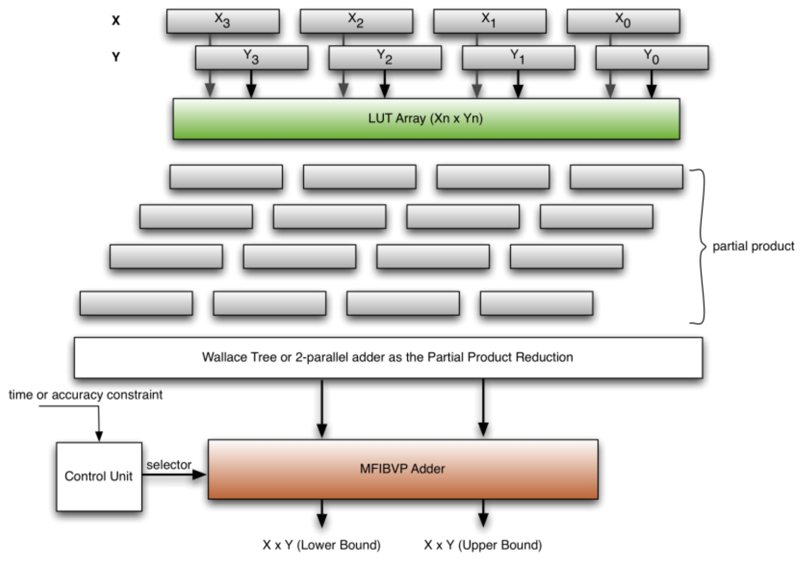


Figure 11 Block diagram of the MFIBVP multiplier.

The second implementation of the multiplication operation depicted in Figure 12 uses the logaritmic-based multiplication. A multiplication of two operands: X and Y by this method performs as the following equation (*b* is the logarithmic base):

$$X \times Y = anti\log_b(\log_b X + \log_b Y) \tag{4}$$

We can use the MFIBVP adder to compute the multiplication so the same features will be produced by this implementation.

We can use look-up table technique [21] or logarithmic schema [23] as the log and antilog blocks shown in Figure 12. The data bus width produced by the log blocks determines the amount of look-up table block in the MFIBVP adder, while the adder's output bus width determines the approximation of the multiplication result produced by the antilog blocks.

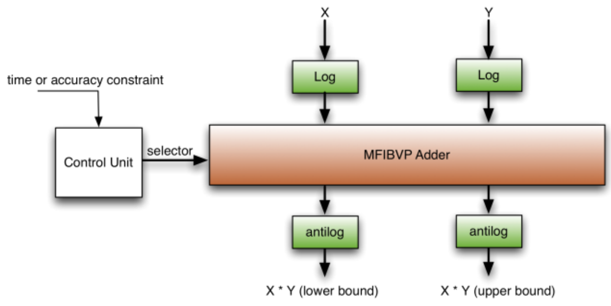


Figure 12 Alternative block diagram of of the MFIBVP multiplier using logarithmic multiplication technique.

3.3 Division

In this paper, the implementation of the division operation uses the same technique as the second approach of the multiplication: logaritmic-based division. A division of two operands X and Y by this method performs as the following equation (*b* is the logarithmic base):

$$X \div Y = anti\log_b(\log_b X - \log_b Y) \tag{5}$$

We can use the MFIBVP subtractor previously mentioned to compute the division so the same features will be produced by this implementation.

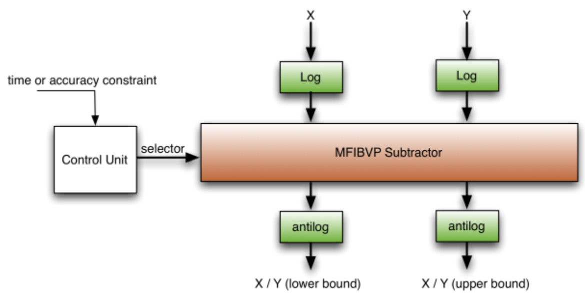


Figure 13 Block diagram of the MFIBVP divisor using logarithmic division scheme.

4 Evaluation of the Proposed Architecture

In this section, we will evaluate the operation of the MFIBVP real-time arithmetic unit. The objective is to determine the time needed by each computation unit in different path delays, and to study the computation accuracy that takes place as a result of the imprecise calculations when processing the incomplete operation. From this evaluation the information necessary will be obtained to establish the suitable clock cycle for the future processor and to relate the available time to the processed part of each function.

4.1 Time Complexity

The time complexity $O(n,k)$ of each MFIBVP real-time operators in this paper is measured as a function of n and k, which represents as the total amount of the gate's delay (Δg) needed.

4.1.1 MFIBVP Real-Time Adder/Subtractor

Based on the block diagram depicted on Figures 8 and 9 the time complexity of the MFIBVP real-time adder/subtractor can be counted by summing the basic or functional gate's delay needed by each steps as Table 1 shows.

Table 1 Time complexity in each step of the n bit MFIBVP real-time k-adder.

Phase	**Step 1 (Most Significant *k*-bit)**	**Step 2**	**Step 3**	**…**	**Final Step**
LUT decoder	K	k	k	...	k
Carry Selector	1	2	3	…	n/k
Multiplexer	$\log_2(n/k)$	$\log_2(n/k)$	$\log_2(n/k)$	…	$\log_2(n/k)$
Total	$k+1+\log_2(n/k)$	$k+2+\log_2(n/k)$	$k+3+\log_2(n/k)$	…	$k+(n/k)+\log_2(n/k)$

4.1.2 MFIBVP Real-Time Multiplier

Based on the block diagram depicted on Figure 11 the time complexity of the MFIBVP real-time multiplier can be counted by summing the basic or functional gate's delay needed by each step as Table 2 shows.

Table 2 Time complexity of the *n* bit MFIBVP real-time *k*-multiplier.

Phase	**Step 1 (Most Significant *k*-bit)**	**Step 2**	**Step 3**	**…**	**Final Step**
Partial product	k	k	k	...	k
Wallace tree	$1+\log_2(n/k)$	$1+\log_2(n/k)$	$1+\log_2(n/k)$	…	$1+\log_2(n/k)$
MFIBVP adder	$k+1+\log_2(n/k)$	$k+2+\log_2(n/k)$	$k+3+\log_2(n/k)$	…	$k+(n/k)+\log_2(n/k)$
Total	$1+1+2k+2\log_2(n/k)$	$2+1+2k+2\log_2(n/k)$	$3+1+2k+2\log_2(n/k)$	…	$(n/k)+1+2k+2\log_2(n/k)$

4.1.3 MFIBVP Real-Time Divider

If the logarithmic precision stored in the LUT ROM for Logarithmic and Antilog block depicted in Figure 13 is *m*-bit, then the time complexity of the MFIBVP real-time divider can be counted by summing the basic or functional gate's delay needed by each step as Table 3 shows.

Table 3 Time complexity of the *n* bit MFIBVP real-time *k*-divider.

Phase	**Step 1 (Most Significant *k*-bit)**	**Step 2**	**Step 3**	**…**	**Final Step**
LOG (LUT decoder)	n	n	n	...	n
MFIBVP substractor	$k+1+\log_2(m/k)$	$k+2+\log_2(m/k)$	$k+3+\log_2(m/k)$	…	$k+(m/k)+\log_2(m/k)$
ANTILOG (LUT decoder)	m	m	m	…	m
Total	$1+n+m+k+2\log_2(m/k)$	$2+n+m+k+2\log_2(m/k)$	$3+n+m+k+2\log_2(m/k)$	…	$(m/k)+n+m+k+2\log_2(m/k)$

4.2 Accuracy Comparison

In this section we will compare the computation accuracy of each architecture of the MFIBVP real-time arithmetic operation previously described with well-known architecture arithmetic operation along with computation time. The accuracy of the MFIBVP real-time operation $\mathrm{A}(t)_{MFIBVP}$ is computed in Equation (6) and Equation (7).

$$A(t)_{comp.MFIBVP} = \mathrm{N}(t)_{MFIBVP.up} - \mathrm{N}(t)_{MFIBVP.low} \quad (6)$$

$$A(t)_{MFIBVP} = \left(1 - \frac{A(t)_{comp.MFIBVP}}{D_{\max}}\right) \times 100\% \tag{7}$$

where:

D_{max} is value between the lowest and highest value that can be produced by particular arithmetic operation,

$\mathrm{N}(t)_{MFIBVP.up}$ and $\mathrm{N}(t)_{MFIBVP.up}$ are upper and lower intermediate-result produced by particular MFIBVP real-time arithmetic operation at t time.

Because of the original nature of the conventional, although well-known, architecture arithmetic operation that only produces single numeric result at the end of operation, we have to assume that they can produce intermediate-result along the computation time thus we can calculate the accuracy of the operation $\mathrm{A}(t)_{\mathrm{conv}}$ as shown in Equation (8) and Equation (9).

$$A(t)_{comp.conv} = \text{Biggest computation value} - \mathrm{N}(t)_{conv} \tag{8}$$

$$A(t)_{conv} = \left(1 - \frac{A(t)_{comp.conv}}{D_{\max}}\right) \times 100\% \tag{9}$$

where:

$\mathrm{N}(t)_{conv}$ are intermediate-result produced by particular conventional arithmetic operation at t time.

4.2.1 MFIBVP Real-Time Adder

By using k-bit as the computation granularity and LUT with both lower and upper computation result stored in ROM, there is novel advantage in the architecture of the MFIBVP real-time adder designed in this research. The advantage is in determining the accuracy of computation regardless the value of both operands. By referring to Figure 6, the difference between the lowest and highest value produced in single LUT block is 1. Based on Equation 1 and Equation 6 we can define the difference between the lowest and highest intermediate-result at computation time t produced by the MFIBVP real-time addition of n bit operands with granularity k:

$$A(t)_{comp.MFIBVP} = \left[\left(\frac{n}{k} - \left\lfloor \frac{t - t_a}{c_{t.MFIBVP}} \right\rfloor\right) @ 1\right]_{base=2^k} \tag{10}$$

The @ symbol represents repetitive value of the right hand side by the value of the left hand side of the @ symbol, for example 4@1 means 1111.

By knowing the accuracy of the intermediate-result produced during computation, based on application-specific computation we can determine its error propagation.

According to Equation (7) and Equation (10), Figure 14 shows the propagation of computation accuracy gained by the MFIBVP real-time adder with *k*=4 compared with the one gained by the Carry Look-ahead Adder and Carry Propagation Adder technique.

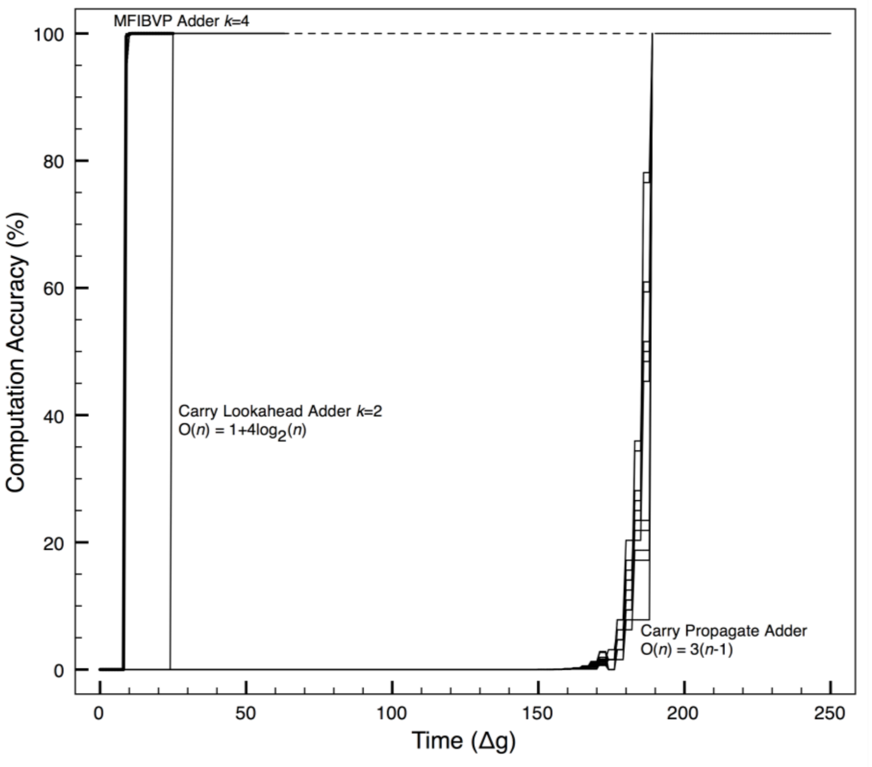


Figure 14 Performance comparison between the MFIBVP real-time adder vs Carry Loo*k*-ahead Adder and Carry Propagate Adder on 50 pairs of 64-bit random numbers.

4.2.2 MFIBVP Real-Time Multiplier

The performance of the MFIBVP real-time multiplier is measured by plotting the accuracy of the intermediate result (Equation (7) and Equation (10)) gained during computation time *t*, compared with the performance of the same multiplier technique (array multiplier) that uses Carry Look-ahead Adder and

Carry Propagate Adder as the final partial product adder. The performance measured in Figure 15 is based on the multiplication of 50 pairs of 32-bit random numbers.

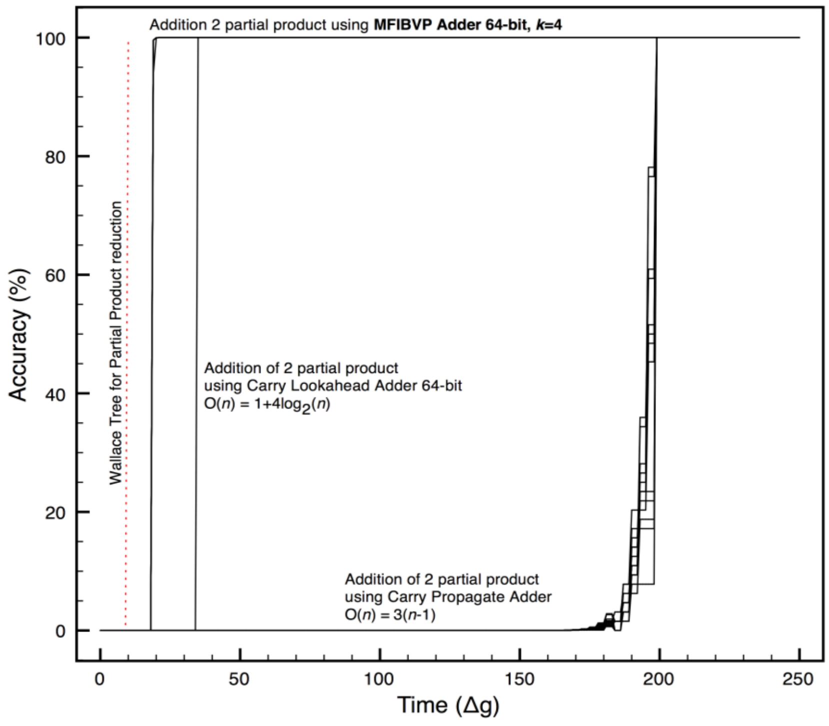


Figure 15 Performance comparison between the MFIBVP real-time multiplier which uses MFIBVP real-time adder as the final partial product adder vs array multiplier that uses Carry Look-ahead Adder and Carry Propagate Adder.

4.2.3 MFIBVP Real-Time Divider

The performance of the MFIBVP real-time divider is measured by plotting the accuracy of the intermediate result (Equation (7) and Equation (10)) gained during computation time *t*, compared with the performance of a well-known division technique: the Carry look-ahead cellular array divider [21] with time complexity $O(n)=(10n+11)\Delta_g$. The performance measured in Figure 16 is based on the division of 50 pairs of 32-bit random numbers.

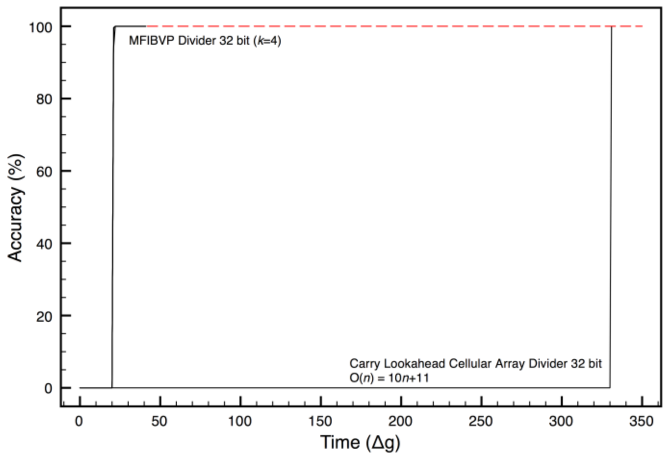


Figure 16 Performance comparison between the MFIBVP real-time divider vs Carry Look-ahead cellular array divider over 50 pairs of 32-bit random numbers.

4.3 Application Example

The development of a real-time arithmetic processor has interesting applications in which the adjustment and determination of the features play a crucial role in the correct computational resolution. This section describes a simple example that illustrates the specific application of the proposed arithmetic unit, the same case as application example presented in [8]. The application consists in controlling the position of object B whose objective is to closely pursue object A. The pursuing object B will have to constantly correct its trajectory in order to adapt to the changes of detected direction in the followed object A. The movement management of B only considers the aspects of tracking A. In the application example, other more distant obstacles or factors are not considered. The determination of the movement is based on the value of the direction tangents of the moving object A with regard to the reference system axes of B. Object A will travel and make correction of its course based on the conventional arithmetic 32-bit computation, and B will try to catch A based on MFIBVP real-time arithmetic technique to compute the angle between B and A in 2D space:

$$\beta_{BA} = \arctan\left(\frac{X_B - X_A}{Y_B - Y_A}\right) \tag{11}$$

Evidently, it is a basic approach; we have simplified the number of variables in order to offer a clear use of the prototype. Figure 17 shows the simplified algorithm in this application to measure the accuracy of MFIBVP real-time arithmetic computation. By changing the computation granularity *p*, we can compare the cycles needed by the object B to catch the object A. Figure 18 shows that starts from the computation with 20 bit of its operand’s width (out of 32) with MFIBVP real-time arithmetic the total cycles needed by object B to catch the object A is just the same with computation with greater precision.

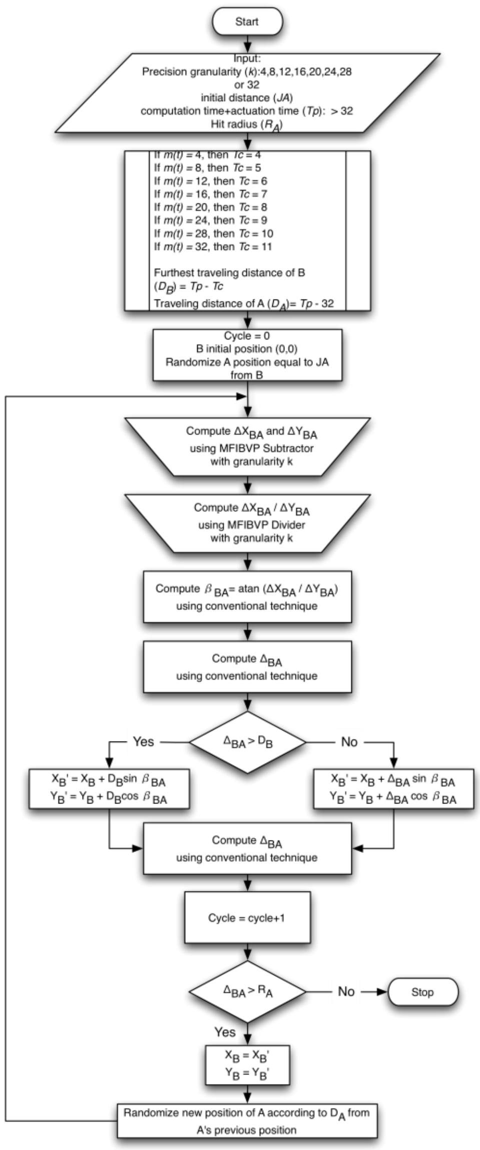

Start
Input:
Precision granularity (k):4,8,12,16,20,24,28
or 32
initial distance (JA)
computation time+actuation time (Tp): > 32
Hit radius (R_A)
If $m(t)$ = 4, then Tc = 4
If $m(t)$ = 8, then Tc = 5
If $m(t)$ = 12, then Tc = 6
If $m(t)$ = 16, then Tc = 7
If $m(t)$ = 20, then Tc = 8
If $m(t)$ = 24, then Tc = 9
If $m(t)$ = 28, then Tc = 10
If $m(t)$ = 32, then Tc = 11
Furthest traveling distance of B
(D_B) = Tp - Tc
Traveling distance of A (D_A)= Tp - 32
Cycle = 0
B initial position (0,0)
Randomize A position equal to JA
from B
Compute ΔX_{BA} and ΔY_{BA}
using MFIBVP Subtractor
with granularity k
Compute $\Delta X_{BA} / \Delta Y_{BA}$
using MFIBVP Divider
with granularity k
Compute β_{BA}= atan ($\Delta X_{BA} / \Delta Y_{BA}$)
using conventional technique
Compute Δ_{BA}
using conventional technique
$\Delta_{BA} > D_B$
Yes
No
$X_B' = X_B + D_B \sin \beta_{BA}$
$Y_B' = Y_B + D_B \cos \beta_{BA}$
$X_B' = X_B + \Delta_{BA} \sin \beta_{BA}$
$Y_B' = Y_B + \Delta_{BA} \cos \beta_{BA}$
Compute Δ_{BA}
using conventional technique
Cycle = cycle+1
$\Delta_{BA} > R_A$
No
Stop
Yes
$X_B = X_B'$
$Y_B = Y_B'$
Randomize new position of A according to D_A from
A's previous position

Figure 17 Simplified algorithm of the object tracking application, the cycle needed by object B to catch object A reflects the accuracy of B’s MFIBVP real-time arithmetic unit.

The accuracy of computation in the object B based on the processed operand's width *p*:

$$A_{(p)} = \frac{\sum cycle_{(p)}}{\sum cycle_{(32)}} \times 100\% \tag{12}$$

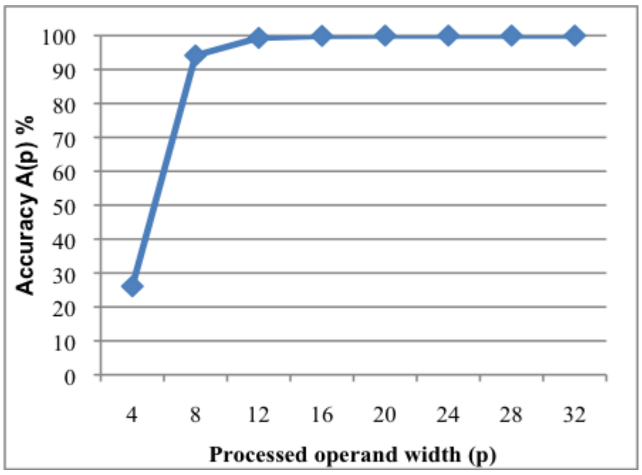


Figure 18 MFIBVP real-time arithmetic performance in the pursuing object application.

5 Conclusion

We can conclude that the MSB-First Interval-Bounded Variable-Precision (MFIBVP) real-time arithmetic unit gives better computation performance by it's ability to:

1. Produce intermediate-result during execution time,
2. Give certainty in computation accuracy even before the process finish time by providing two intermediate-results which act as the lower and upper bound of the real and complete computation result,
3. Gain high computation accuracy from the early time of the execution process.

Nomenclature

$\otimes$	=	arithmetic operations
$A(t)_{ars}$	=	accuracy of the operation by the by the *ars* arithmetic unit at *t* time of computation
$A(t)_{comp.ars}$	=	difference between the upper and lower numeric value produced by the *ars* arithmetic unit at *t* time of computation

$c_{t.ars}$	=	time constant to process single pair of operands' bit needed by the *ars* arithmetic unit
D_{max}	=	difference between the maximum and minimum numeric value possible on a particular arithmetic operation
K	=	operation's granularity
$m(t)$	=	total bits that have been compute as a function of time
n	=	operand's width
$\mathrm{N}(t)_{ars}$	=	numeric value of the arithmetic computation carried out by the *ars* arithmetic unit at *t* time of computation
t_a	=	start time of the arithmetic execution of the *ars* arithmetic unit
$t_{f.ars}$	=	finish time of the arithmetic execution of the *ars* arithmetic unit
$T_{p.ars}$	=	period of the process time needed by the *ars* arithmetic unit to finish its arithmetic operation

References

[1] Laplante, P.A., *Real-Time Systems Design and Analysis*, IEEE Press Wiley Interscience, 2004.

[2] Zadeh, L.A., *Soft Computing and Fuzzy Logic*, IEEE Software, **11**, 48-56, Nov. 1994,

[3] Liu, J.W.S., Lin, K., Shih, W. & Yu, A.C., *Algorithms for Scheduling Imprecise Computations, Computer*, **24**, 56-68, May. 1991.

[4] Lim, C.C. & Zhao, W., *Performance analysis of dynamic multitasking imprecise computation system*, IEE Proceedings of Computers and Digital Techniques, **138**, pp. 345-350, Sep. 1991.

[5] Liu, J.W.S., Shih, W.S., Lin, K., Bettati, R. & Chung, J., *Imprecise Computations,* Proceedings of the IEEE, **82**, Jan. 1994.

[6] Huang, X. & Cheng, A.M.K., *Applying imprecise algorithms to real-time image and video transmission*, Proceedings of the Real-Time Technology and Applications Symposium, pp. 96, 1995.

[7] Shih, W. & Liu, J.W.S., *Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error,* IEEE Transactions on Computers, **44,** 466, , Mar. 1995.

[8] Mora-Mora, H., Mora-Pascual, J., Garcia-Chamizo, J.M. & Jimeno-Morenilla, A., *Real-time arithmetic unit,* Real-Time Systems, **34**, 53-79, 2006.

[9] Kuspriyanto & Kerlooza, Y.Y., *Multiple Operand Real-Time Adder*, Proceeding of SIK 2003, 2003.

[10] Kuspriyanto & Kerlooza, Y.Y., *Menuju Prosesor Waktu-Nyata: Dual Algorithms Real-Time Adder*, Prosiding Seminar on Electrical Engineering (SEE), Univ. Ahmad Dahlan Yogyakarta, 2003.

[11] Kuspriyanto & Kerlooza, Y.Y., *Keandalan Unit Multioperand MSB-First Real-Time Adder Pada Operasi Penjumlahan Data Acak*, Proceeding: Seminar on Intelligent Technology and Its Applications 2004 (SITIA'2004), 2004.

[12] Kuspriyanto & Kerlooza, Y.Y., *Towards New Real-Time Processor: The Multioperand MSB-First Real-Time Adder*, Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04), pp. 524 529, Aug. 2004.

[13] Kerlooza, Y.Y. & Kuspriyanto, *Towards Real-Time Processor: Multioperand MSB-First Minimax Addition*, International Conference on Electrical Engineering and Informatics (ICEEI2007), 2007.

[14] Kerlooza, Y.Y. & Kuspriyanto, *Towards Real-Time Processor: The Implementation of Multioperand MSB-First Adder Arithmetic Unit on the Computation of $y = \Sigma$ ai bi*, International Conference on Electrical Engineering and Informatics (ICEEI2007), Jun. 2007.

[15] Kerlooza, Y.Y. & Kuspriyanto, *Real-Time dan Adjustable Computing*, e-Indonesia Initiative 2008 (eII2008), May. 2008.

[16] Nielsen, A.M. & Kornerup, P., *MSB-First Digit Serial Arithmetic,* Journal of Universal Computer Science, **1**, 1995.

[17] Moore, R.E. & Yang, C.T., *Interval analysis I*, Lockheed Missiles and Space Co., 1959.

[18] Moore, R.E., *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Department of Mathematics, Stanford University, Stanford, California, 1962.

[19] Boche, R.E., *An Operational Interval Arithmetic*, IEEE-Illinois Inst. of Tech.-Northwestern Univ., Univ. of Illinois. Abstract of a paper given at National Electronics Conference, 1963.

[20] Deng, Z. & Liu, J.W., *Scheduling real-time applications in an open environment*, in Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer, Society Press, pp. 308–319, 1997.

[21] Lu, Mi., *Arithmetic and Logic in Computer Systems*, John Wiley & Sons Inc., 2004.

[22] Wallace C.S., *A Suggestion for a Fast Multiplier*, IEEE Transaction on Computers, **13**, 14-17, 1964.

[23] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, USA, 1999.