ETSI TR 126 973 V15.1.0 (2018-10)

TECHNICAL REPORT

5G;
Update to fixed-point basic operators
(3GPP TR 26.973 version 15.1.0 Release 15)

Reference

RTR/TSGS-0426973vf10

Keywords

5G

Intellectual Property Rights

Foreword

This Technical Report (TR) has been produced by ETSI 3rd Generation Partnership Project (3GPP).

The present document may refer to technical specifications or reports using their 3GPP identities, UMTS identities or GSM identities. These should be interpreted as being references to the corresponding ETSI deliverables.

The cross reference between GSM, UMTS, 3GPP and ETSI identities can be found under http://webapp.etsi.org/key/queryform.asp.

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Contents

Foreword

This Technical Report has been produced by the 3rd Generation Partnership Project (3GPP).

The contents of the present document are subject to continuing work within the TSG and may change following formal TSG approval. Should the TSG modify the contents of the present document, it will be re-released by the TSG with an identifying change of release date and an increase in version number as follows:

Version x.y.z

where:

x the first digit:

1 presented to TSG for information;

2 presented to TSG for approval;

3 or greater indicates TSG approved document under change control.

y the second digit is incremented for all changes of substance, i.e. technical enhancements, corrections, updates, etc.

z the third digit is incremented when editorial only changes have been incorporated in the document.

Introduction

The last major update to the ITU-T Basic Operators [6] was in 2005, with a follow on update in 2009. These basic operators serve as a foundation for reference software of codecs specified by 3GPP. During the last several years, processors with wide accumulators, and support for single-instruction-multiple-data (SIMD), and very long instruction word (VLIW) features have become prevalent. The basic operators of 2009 now need to be extended to leverage these capabilities of modern processors so that implementations with lower mega-cycles-per-second (MCPS) and lower-power may be realized.

Enhanced Voice Services (EVS) is one of the recent codecs defined by 3GPP that can leverage these features of modern processors. The existing EVS reference software would have to be appropriately modified to leverage these extended basic operators without changing the underlying algorithm. This is referred to as an alternative EVS implementation using the extended basic operators.

This alternative EVS implementation would have to be evaluated to ensure that inter-operability is maintained in addition to ensuring that voice quality is not impacted.

1 Scope

The present document covers the following topics:

1) Assessment of the gaps between modern processors and the existing set of basic operators (STL2009) [6].
2) Proposal of an extended set of operators addressing modern DSP architectures as an extension to STL2009.
3) Assessment of merits of an alternative EVS implementation using extended STL2009 Basic Operators.
4) Proposal for validation of an alternative EVS implementation using extended STL2009 Basic Operators.

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication, edition number, version number, etc.) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies. In the case of a reference to a 3GPP document (including a GSM document), a non-specific reference implicitly refers to the latest version of that document *in the same Release as the present document*.

[1] 3GPP TR 21.905: "Vocabulary for 3GPP Specifications".

[2] 3GPP TS 26.442: "Codec for Enhanced Voice Services (EVS); ANSI C code (fixed-point)".

[3] Recommendation ITU-T P.800 (08/1996): "Methods for subjective determination of transmission quality".

[4] Recommendation ITU-T P.863 (09/2014): "Perceptual objective listening quality assessment".

[5] 3GPP TS 26.443: "Codec for Enhanced Voice Services (EVS); ANSI C code (floating-point)".

[6] Recommendation ITU-T G.191 (03/10): "Software tools for speech and audio coding standardization".

[7] 3GPP TR 26.952: "Codec for Enhanced Voice Services (EVS); Performance Characterization (Release 14)".

3 Abbreviations

For the purposes of the present document, the abbreviations given in 3GPP TR 21.905 [1] and the following apply. An abbreviation defined in the present document takes precedence over the definition of the same abbreviation, if any, in 3GPP TR 21.905 [1].

SIMD	Single Instruction Multiple Data
STL	Software tools for speech and audio coding standardization
VLIW	Very Long Instruction Word.

4 Extension to the STL2009 Basic Operators

4.1 Analysis of the gap between current basic operators and modern DSP architectures

State-of-the-art processor architectures, such as the recent ones from Intel, ARM, QUALCOMM, Texas Instruments etc., support wide accumulators, SIMD and VLIW capabilities. The last major update to the ITU-T Basic Operators was in 2005, with a follow on update in 2009 [6]. It appears that these earlier versions of the Basic Operators (2009 and earlier) were influenced by older DSP architectures such as the Texas Instruments TMS320C5x and TMS320C54x processors where the accumulator was 40 bits wide.

However, a survey of the state-of-the-art processor architectures shows that most of them support the following capabilities:

- Wider (64 bit) accumulators and registers.
- Wider accumulators enable additional guard bits which eliminate the need for checking for saturation after every basic operation.
- SIMD (Single Instruction Multiple Data) instructions which can process vector data. For example, a single instruction can process two 32-bit data elements or four 16-bit elements in parallel.
- VLIW (Very Long Instruction Word) enables several operations to be executed in parallel in a single cycle.

Basic operators that are friendlier to compilers, and enable SIMD and VLIW features to be leveraged, can significantly reduce implementation time. Improved compiler technology and software development tools interpret data types and associated basic operators to map them to a processor architecture for better Out-of-box (OOB) performance. Without this computer assisted optimization, an engineer would have to hand-optimize the code which would result in increased engineering effort and longer time to market.

Many recent audio/hybrid codecs make extensive use of 16bit x 32bit MAC (multiply and accumulate) and 32bit x 32bit MAC operations which are realized quite differently between VLIW and SIMD architectures and the current Basic Operators:

- Current STL2009 Basic operators require saturation and truncation after every multiply-accumulate (MAC) operation to maintain bit-exactness.
- The current Basic operator saturation checks prevent use of SIMD parallelism.
- To maintain bit-exactness, cycles are wasted resulting in higher MCPS and power on VLIW and SIMD capable devices.
- Higher precision variables, such as 64bit operands, are partitioned into smaller width operands, processed and then put back to the original width. This results in an overhead and processor cycles are wasted.

Considering the capabilities of modern processor architectures, as well as the characteristics of the latest speech and audio codecs, there is a need for extending STL2009 with additional basic operators & data types to better leverage the capabilities of state-of-the-art processor architectures and characteristics of DSP algorithms.

4.2 Test methodology for validating the extended basic operators

4.2.0 General

This clause describes a test framework that will compare the fixed-point arithmetic accuracy of the extended basic operators against a floating-point implementation of the extended basic operators. Each basic operator will be tested for 4 different data patterns.

In table 1 below, the extended basic operators have been classified into four main classes. The test patterns used for testing and the build options of the test framework are also shown below.

Table 1: Classification of the extended basic operators

Test framework for extended basic operators			
Main class	**Subclass**	**Total basops**	**Covered basops**
64 bit accumulator	64-bit Integer Mac	4	4
	64-bit Mac	7	7
	64-bit Math	12	12
	64-bit scale	7	7
	64-bit move	5	0
Complex	Complex Math	7	7
	Complex Mac	9	9
	Complex Move	10	0
	Complex Scale	9	9
Enhanced 32 bit	32*16 bit Enh MAC	6	6
	32*32 bit Enh MAC	6	6
Control code ops		18	0
Total		100	67

Test data patterns:

- -1.0 to 1.0 float range with configurable interval.
- Random numbers.
- Special values: very low level values (e.g., in the range of 1e-3, 1e-6 etc.), nominal and large values
- Custom mode: users can specify their customized array of size N.

Build options:

MSVC 2017 and MSVC 2013 workspaces are provided, with 2 options:

- MSVC 2017/2013 project.
- Gcc based makefiles.

4.2.1 Test methodology

In Figure 1 below, a block diagram explains how to validate the extended STL2009 Basic Operators implementation against a reference floating-point implementation. A data generator generates floating-point notation data values that are then converted into fixed-point notation and these are input to the design under test (DUT) implementation of the extended STL2009 Basic Operators implementation. The same fixed-point data is converted into floating-point notation, and then input to a reference floating-point implementation of the extended STL2009 Basic Operators. The fixed-point output of the DUT is converted to floating-point notation, and then compared against the reference floating-point implementation output and an error value is generated and logged.

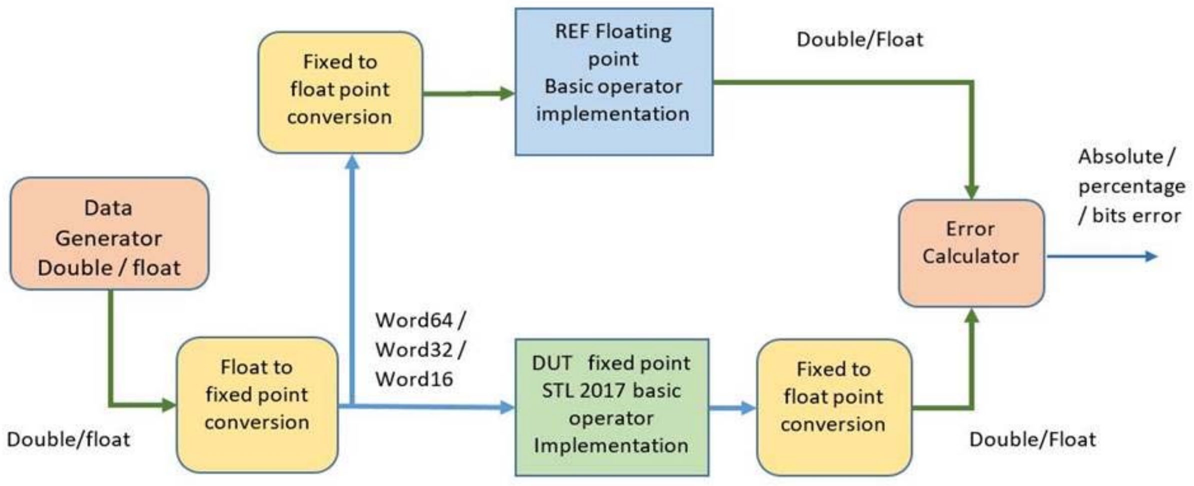
Fixed to float point conversion
REF Floating point Basic operator implementation
Double/Float
Absolute / percentage / bits error
Data Generator Double / float
Error Calculator
Word64 / Word32 / Word16
Float to fixed point conversion
DUT fixed point STL 2017 basic operator Implementation
Fixed to float point conversion
Double/float
Double/Float

Figure 1: Block diagram illustrating how the fixed-point implementation is validated against a floating-point reference implementation of the extended STL2009 basic operators

In the following clauses, the test results for an example basic operator, Mpy_32_16_1 are reported.

4.2.2 Test results for basic operator Mpy_32_16_1

The setup in figure 1 was used for testing with four different types of data:

1) Random input numbers
2) A sweep from a negative number to a positive number
3) A piecewise sweep from a negative number to a positive number
4) A custom input where a user can specify an array of size N with custom inputs

Figures 2, 3, 4 and 5 illustrate the results of the test for the above four different data types. The error between the fixed-point implementation and floating-point implementation are extremely small thereby validating the fixed-point implementation.

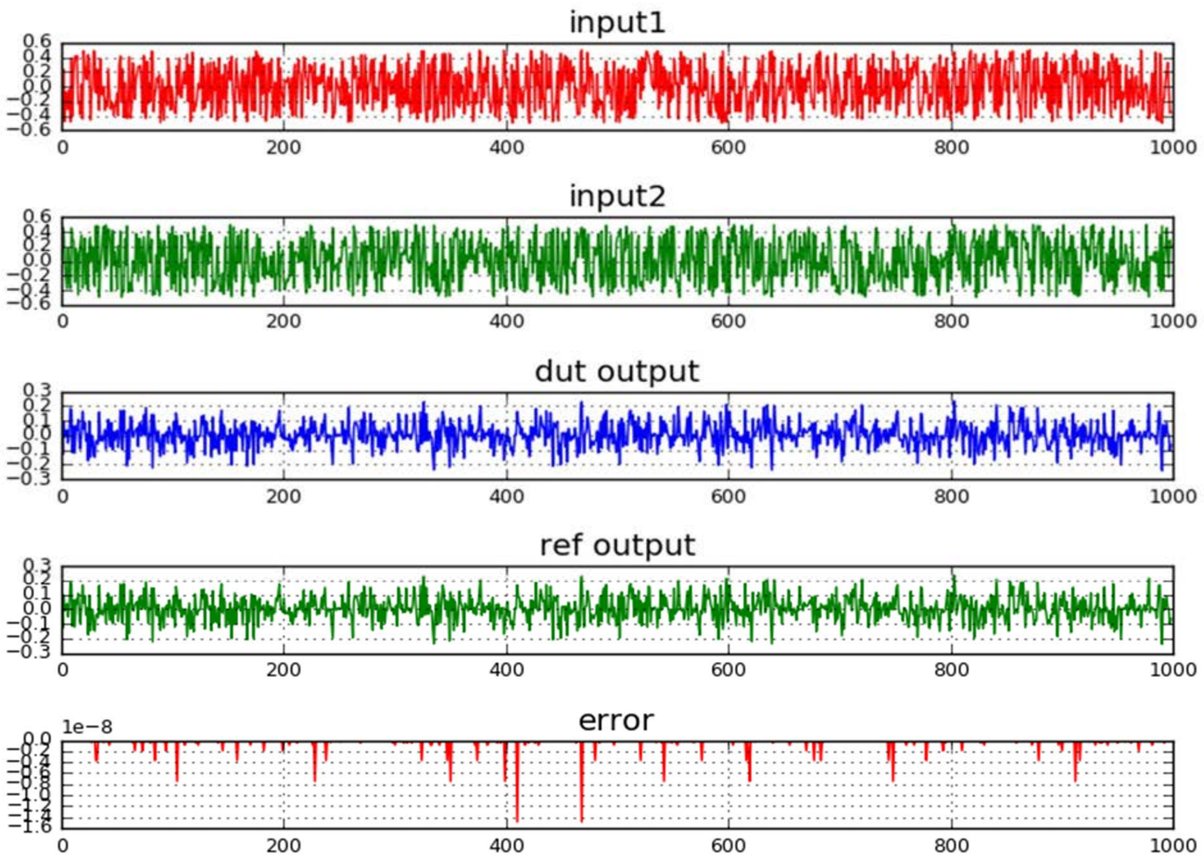

Figure 2: Test results for basic operator Mpy_32_16_1 using random input. The error between the fixed-point output and floating-point output is very small.

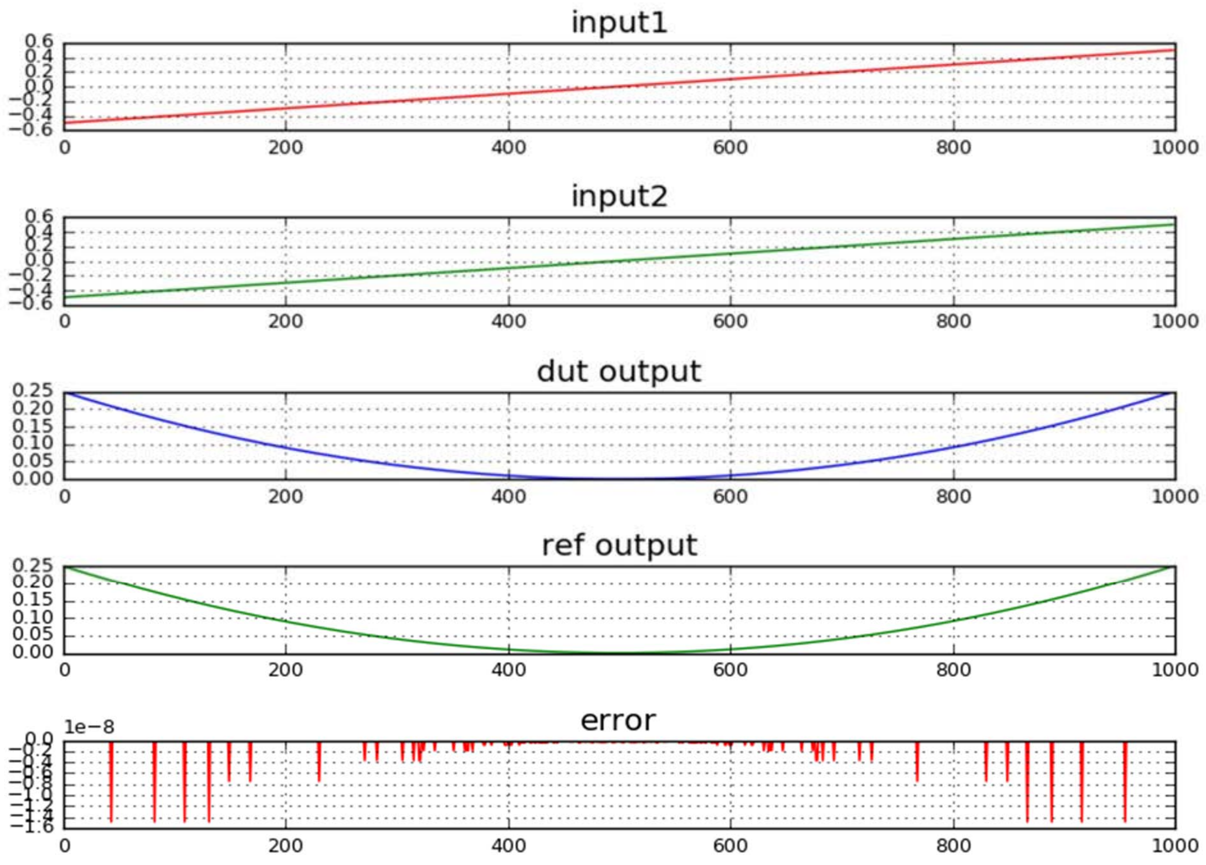

Figure 3: Test results for basic operator Mpy_32_16_1 using a sweep input. The error between the fixed-point output and floating-point output is very small.

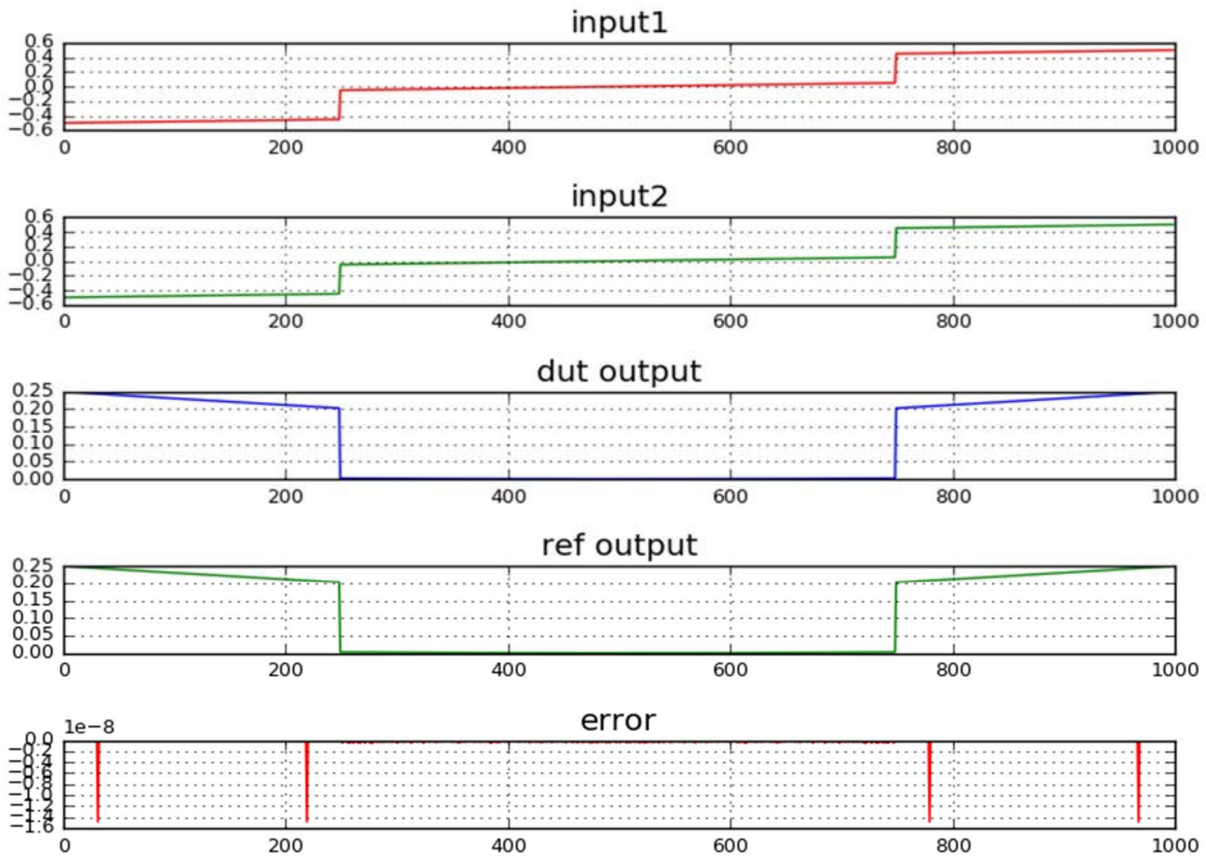

Figure 4: Test results for basic operator Mpy_32_16_1 using a piecewise sweep input. The error between the fixed-point output and floating-point output is very small.

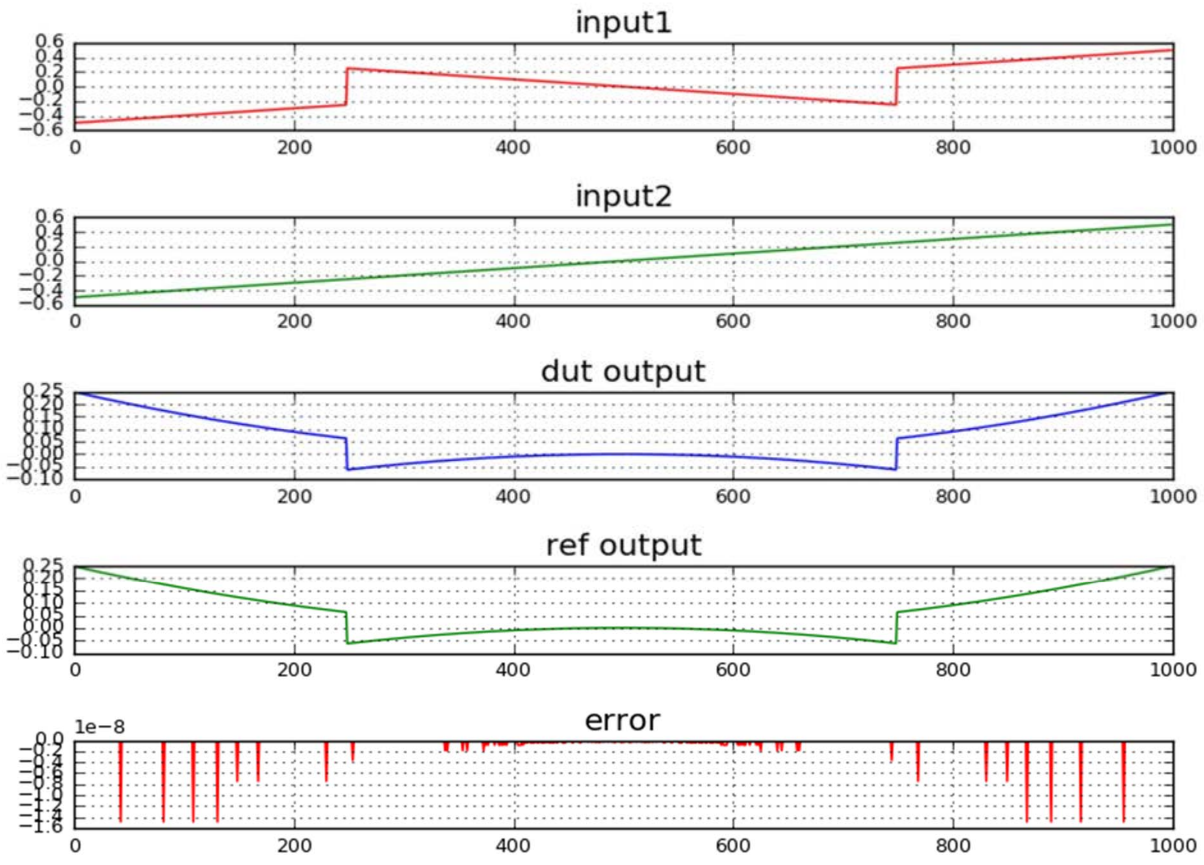

Figure 5: Test results for basic operator Mpy_32_16_1 using a user defined custom input. The error between the fixed-point output and floating-point output is very small.

4.2.3 Test results

For a complete report of the framework used, as well as the results of the test, please see the attachment "Baseop_tst_frmwork.zip".

NOTE: The unsigned basic operators in clause A.5 were verified separately and are used by the EVS codec in TS 26.442 [2].

4.2.4 Test results conclusion

Based on the results reported in "precision_abs_err_report.csv", it can be concluded that the fixed-point implementation of the extended basic operators all pass against the reference floating-point implementation of the same extended basic operators.

5 Alternative EVS Implementation Using the Extended Basic Operators

5.1 Merits of an alternative EVS implementation using the extended basic operators

EVS [2] is a sophisticated hybrid audio-speech codec with several modes of operation. As such it has a large number of functions. Manually optimizing this large set of functions is prohibitive from an effort (and therefore time) perspective. Implementers will have to rely on computer assisted tools and compiler to get them as close to a final implementation as possible, and spend the last mile in manual optimization to reach the final target performance. It is therefore imperative that the basic operators are defined in such a manner that they lend themselves to better leverage the features and capabilities of modern DSP architectures. Data types need to be mapped to match the processor registers or operand widths of data used in SIMD (Single Instruction Multiple Data) processing; basic operators need to be mapped to processor instructions. A standard reference C code written with these aspects in mind will result in an implementation that leverages SIMD and VLIW (Very Long Instruction Word) features of the processor better and results in an out-of-the-box (OOB) performance that is quite close to the final desired performance. The compiler can optimize the code across all the files and functions thereby significantly reducing manual optimization effort. Implementers can go to market faster.

Figure 6 shows the benefits of creating an alternate reference C code for EVS using the updated basic operator:

1) Reduced hand-optimization efforts lead to reduced total engineering effort, and hence improved time to market.

2) Improved MCPS numbers in OOB and final hand-optimized code.

3) Reduced code size. Reduced MCPS and memory reduces overall power used. This should facilitate extended battery life.

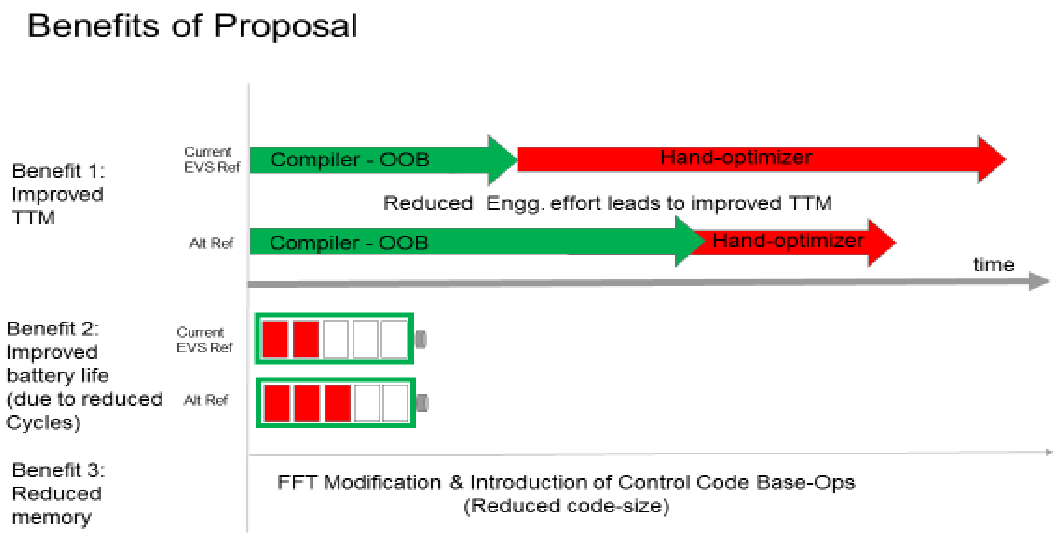
Benefits of Proposal
Benefit 1: Improved TTM
Current EVS Ref
Compiler - OOB
Hand-optimizer
Reduced Engg. effort leads to improved TTM
Alt Ref
Compiler - OOB
Hand-optimizer
time
Benefit 2: Improved battery life (due to reduced Cycles)
Current EVS Ref
Alt Ref
Benefit 3: Reduced memory
FFT Modification & Introduction of Control Code Base-Ops (Reduced code-size)

Figure 6: Benefits of proposed alternate reference C for EVS

Using the existing standard EVS Reference code version 14.0.0 as a starting point, an alternative C code that leverages the proposed basic operators has been created. During this creation process, step by step, several key parameters have been monitored such as the engineering effort spent expressed as time (days, weeks, months), and corresponding reduction in MCPS.

Figure 7 shows the optimization level achieved versus engineering effort measured in units of time. As the figure shows, the OOB performance of the existing reference C is at 269 MCPS, while the OOB performance of the proposed alternative EVS reference C code is at 162 MCPS. This is a gain of 1.66x achieved in matter of a few days of engineering effort. Next, time is spent restructuring the code and hand optimizing. The final hand-optimized version is at 61.9 MCPS compared to 77.5 MCPS for the existing EVS reference implementation. This is a gain of 1.25x.

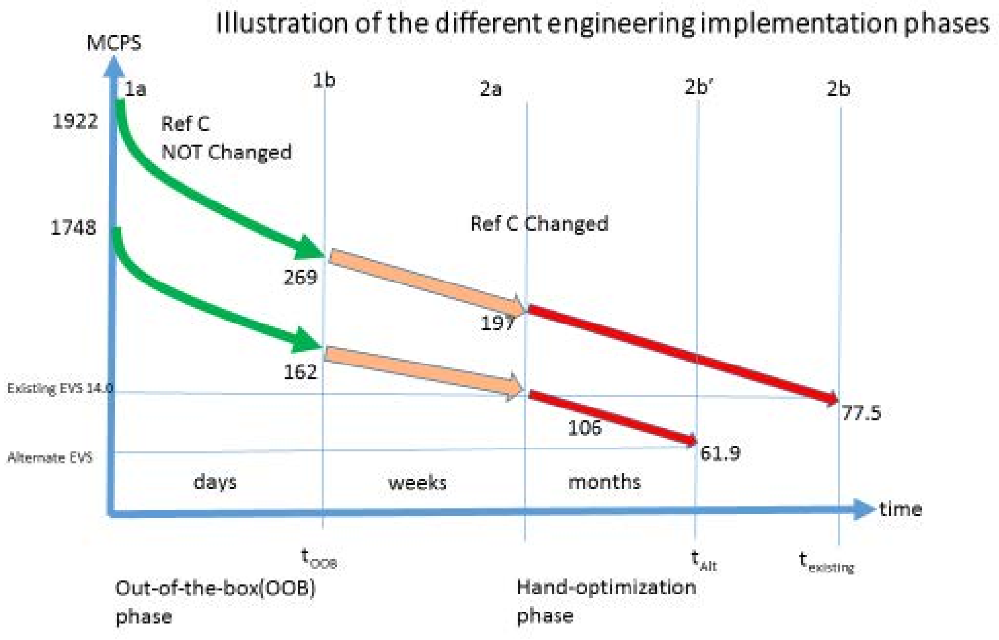

Figure 7: Impact of alternate reference C at different phases of the implementation process

In table 2, the improvement in weighted million operations per second (WMOPS) of the alternative EVS implementation using extended basic operators is compared against the WMOPS of the existing EVS standard reference code using STL2009 basic operators as a baseline. Second row shows a benefit of 1.07x with changing the weights for STL2009 basic operators. Third row shows the total benefit of 1.17x with the use of the extended basic operators and weight change of the existing STL2009 basic operators.

Table 2: WMOPS based Comparison of the alternative EVS implementation with existing EVS implementation

EVS Code Base - 14.0.0	STL_basops complexity weights	Average WMOPS			Improvement Over Reference
		Encoder	Decoder	Total	
Reference with STL2009	STL2009 weights as is	53.3	24.2	77.5	1.00x
Reference with STL2009	With new proposed weights for STL2009	50.6	22.1	72.7	1.07x
Alternate Reference with STL2017	With new proposed weights for STL2009 & for extended basic operators	47.1	18.9	66	1.17x

Following test cases were used for WMOPS and MCPS calculation:

- Encoder test case: -rf HI 3 13200 32 stv32n2.INP stv32n2_rfHI3_13200_32kHz.COD
- Decoder test case: 32 stv32c_rfHI3_13200_32kHz.COD stv32c_rfHI3_13200.out

The WMOPS numbers reported in Table1 are average WMOPS for this worst case complexity test vector. Please refer to 3GPP TR 26.952: Codec for Enhanced Voice Services (EVS); Performance Characterization (Release 14) [7], for a more detailed explanation of WMOPs for EVS.

In table 3, the improvement in million cycles per seconds (MCPS) of the alternative EVS implementation is compared against the MCPS of the existing EVS standard reference code on a specific DSP platform using STL2009 basic operators as a baseline. A gain of 1.25x is observed.

The gain in final MCPS of 1.25x is significantly more than gain of 1.17x in WMOPS. The explanation is that the existing method of computing WMOPS does not address the cycles gained with VLIW where multiple instructions are executed in parallel. In addition, the current assigned integer weights of 1 or higher for SIMD and VLIW friendly instructions does not account for the inherent parallelism possible of processing multiple operands in a single cycle in modern processors.

Table 3: MCPS based Comparison of the alternative EVS implementation with existing EVS implementation on a Cadence Tensilica HiFi DSP

Perf parameter	REFC with STL2009	ALT_REFC with STL2017	Performance improvement
	Total (Enc + Dec)	Total (Enc + Dec)	
OOB MCPS	269.3	162.5	1.66x
Final MCPS	77.5	61.9	1.25x
Code size – OOB (in K Bytes)	2117.3	2036.6	1.04x

5.2 Example pseudo code to illustrate some of the benefits of modern DSP architectures

The following examples illustrate the benefits of VLIW and SIMD features of modern DSP architectures. The existing reference code needs to be changed to leverage the extended basic operators that exploit the features of modern DSP architectures. The following examples with pseudo code show that cycles are reduced from 4 to 2.

Example 1:

Original Reference C Code –

```
for (i=0: i<N; i++)

{

    acc = acc + a[i]*b[i];  /* multiply, truncate, and saturate are happening */

}
```

/* Regular implementation */

/* Multiply, **truncate**, and **saturate** are happening for each element. */

/* Truncate and saturate here imply that order of execution is important. Compiler cannot change this order of execution without violating bit-exactness */

```
Int_32 acc;

acc = a[0]*b[0]; /* cycle 1 */

acc = acc + a[1]*b[1]; /* cycle 2 */

acc = acc + a[2]*b[2]; /* cycle 3 */

acc = acc + a[3]*b[3]; /* cycle 4 */
```

/* total cycles = 4: For processing 4 elements of array a and b */

/* For N elements it will take N cycles */

Example 2:

Explanation of:

- How SIMD/VLIW friendly REFC code helps to reduce cycles.
- Why bit-exactness is violated when VLIW, SIMD features are used.

/* **Example 2 - A:** Implementation in 2 slots VLIW architecture */.

/* Since truncation and saturation is not required, Acc1 and Acc2 executed in 1 cycle in two different slots */

/* Final result in Acc does NOT match acc in regular implementation */

/* This 2-slot implementation is **not bit-exact with regular implementation** and therefore the need to define alternate set of bit-streams */

/* Therefore the reference code has to be changed to take benefit of 2-slot architecture */

Int_64 Acc1, Acc2, Acc;

Acc1 = a[0]*b[0]; /* slot 0, cycle 1 */

Acc2 = a[1]*b[1]; /* slot 1, cycle 1 */

Acc1 = Acc1 + a[2]*b[2]; /* slot 0, cycle 2 */

Acc2 = Acc2 + a[3]*b[3]; /* slot 1, cycle 2 with VLIW supported */ /* Alternatively, this can be slot 0, cycle 2 if 2-way SIMD is supported as illustrated in Example 2-B */

Acc = Acc1 + Acc2; /* slot 0, cycle 3. This will be done outside the loop, only once */

/* Total cycles for 4 elements = 3 */

/* For N elements it will take (N/2 + 1) cycles */

/* **Example 2 - B:** Implementation in 2 slots VLIW and 2-way SIMD architecture */.

/* Since truncation and saturation is not required, Acc1 and Acc2 executed in 1 cycle in two different slots */

/* In a 2-way SIMD architecture, 2 MAC operations can be done in a single cycle in single slot on two-32bit elements stored in a 64 bit registers */

/* This SIMD/VLIW implementation is **not bit-exact with regular implementation** and therefore the need to define alternate set of bit-streams */

/* Therefore the reference code has to be changed to take benefit of 2-way SIMD and 2-slot architecture */

Int_64 Acc1, Acc2, Acc;

/* One 64-bit register holds two 32 bit elements a[0] and a[1]. Another 64-bit register holds two 32 bit elements b[0] and b[1]*/

Acc1 = a[0]*b[0] + a[1]*b[1]; /* slot 0, cycle 1 2-way SIMD mac */

Acc2 = a[2]*b[2] + a[3]*b[3]; /* slot 1, cycle 1 2-way SIMD mac*/

Acc = Acc1 + Acc2; /* slot 0, cycle 2. This will be done outside the loop, only once */

/* Total cycles for 4 elements = 2 */

/* For N elements it will take (N/4 + 1) cycles */

In conclusion, for a loop processing N elements,

Example 1 will consume: N Cycles.

Example 2A will consume: (N/2 + 1) cycles.

Example 2B will consume: (N/4 + 1) cycles.

Hence a 2-way SIMD, 2-slot architecture, will provide close to 4X improvement in cycles for operations in a loop as shown above.

5.3 Validation of an alternative EVS implementation using updated basic operators

5.3.1 C-code inspection

Before starting the performance evaluation, the C-code of the alternative EVS implementation will be shared for inspection, upon request, under NDA. The verification will be reported to SA4.

5.3.2 Objective performance evaluation of the alternative EVS implementation

For the objective performance validation of the alternative implementation of EVS using the updated set of basic operators, it is proposed to use the same procedure as has been used to validate the EVS floating-point. Namely, it is proposed to process a P.800 compatible database [3] [exact database tbd] including speech and music and mixed test samples by the following 4 combinations of the legacy fixed-point EVS [2] encoder and decoder (Ref_fxd) and the evaluated EVS encoder and decoder (CuT):

a) Ref_fxd encoder – Ref_fxd decoder

b) CuT encoder – CuT decoder

c) Ref_fxd encoder – CuT decoder

d) CuT encoder – Ref_fxd decoder

The processing is performed according to EVS-7c and the resulting stimuli are evaluated using POLQA [4] with the reference item being the direct item of the respective bandwidth and the test items being the EVS conditions. In other words all stimuli are evaluated against the original signal.

For each condition and for each P.800 sample, the individual POLQA MOS-LQO scores are computed and the differences for [a) – b)], [a) – c)] and [a) – d)] compared, both for the samples individually, and averaged for each test condition. The proposed alternative EVS implementation and the standardized fixed-point implementation are considered to perform equivalent if the difference values are within reasonable bounds.

It is further proposed to also objectively validate the performance of interoperation of this new EVS implementation with the standardized EVS floating-point implementation [5] (Ref_flt) to make sure that there are no interoperability issues when interoperating with the standardized floating-point EVS code. Consequently, two additional combinations are added:

e) Ref_flt encoder – CuT decoder

f) CuT encoder – Ref_flt decoder

It is proposed that the objective evaluation is performed for all the conditions that were subjectively evaluated in the EVS Selection Tests and for all conditions that were subjectively evaluated in the EVS Characterization Tests.

The analysis would follow the template in Table 4 for all the individual samples and all conditions. Additionally, for each test condition, as well as for all the conditions combined, the following statistics will be also provided – average difference, minimum difference, maximum difference, standard deviation and 95% confidence interval. For better visualization, histograms or cumulative distribution functions of the differences may also be provided.

Table 4: Template for result presentation

Input	Bandwidth	Bit rate	DTX	Level	FER/Profile	a) - b)	a) - c)	a) – d)	a) – e)	a) – f)

5.3.3 Subjective performance evaluation of the alternative EVS implementation

The goal of the subjective performance evaluation of the alternative EVS implementation is to complement the objective validation as a sanity check. It covers all relevant configurations with emphasis on most relevant ones to minimize the number of subjective tests. In particular:

1) Bitrates: All EVS bitrates are included, both of the EVS native modes (5.9, 7.2, 8, 13.2, 13.2 CAM, 16.4, 24.4, 32, 48, 56, 96, 128 kb/s) and the AMR-WB IO modes (23.85, 23.05, 19.85, 18.25, 15.85, 14.25, 12.65, 8.85 and 6.6 kb/s). This is done through constant bitrate conditions or bitrate switching conditions in order to minimize the necessary number of subjective experiments, and yet cover all the bitrates.

2) Bandwidth: It is proposed to include only WB and SWB experiments in the subjective evaluation as most relevant for EVS operation. Further, it is assumed that most of the NB technologies are also included within WB or SWB EVS operation. Finally FB operation is algorithmically very similar to the SWB operation.

3) Input levels: 16, 26, 36 dBov input levels are tested.

4) Noisy speech is evaluated in one experiment.

5) Mixed & Music inputs are evaluated in one experiment.

6) Impaired channel & Jitter Buffer Management (JBM) conditions are spread across all experiments. The Frame Erasure Rates (FERs) or network error profiles have been selected such that they should allow to uncover any issues in operation in impaired channels, yet the channel is not too bad to significantly influence the test resolution for clean channel conditions.

7) Rate switching is included, as mentioned above.

8) Tandem conditions were not included in the test as it is assumed that any implementation issues should be uncovered in conditions without tandeming. Further, tandem operation is not foreseen as a major operation use-case for EVS.

The methodology used is P.800 ACR or DCR reflecting the EVS Selection and Characterization tests. It is proposed to use 4 different talkers (two male and two female talkers), and 6 panels of 4 listeners. This set-up gives 96 votes per condition (6panels*4talkers*4listeners).

Similarly to the objective tests, the following 4 configurations will be tested in all experiments:

a) Ref_fxd encoder – Ref_fxd decoder

b) CuT encoder – CuT decoder

c) Ref_fxd encoder – CuT decoder

d) CuT encoder – Ref_fxd decoder

Experiment 1 - WB clean speech ACR (17 conditions per codec configuration):

-16 dBov clean channel - 5.9 kb/s, switching: 7.2-9.6 kb/s, 13.2-96 kb/s, AMR-WB IO, DTX ON

-26 dBov clean channel - 7.2 kb/s, 13.2 kb/s, 13.2 kb/s Channel-Aware Mode (CAM), 24.4 kb/s, DTX ON

-36 dBov clean channel - 5.9 kb/s, switching: 7.2-9.6 kb/s, 13.2-96 kb/s, AMR-WB IO, DTX OFF

-26 dBov random 3% FER - 5.9 kb/s, switching: 7.2-9.6 kb/s, 13.2-96 kb/s, AMR-WB IO, DTX ON

-26 dBov Profile 8(6.2%) – 13.2 kb/s Channel-Aware Mode (CAM), DTX ON

Experiment 2 - SWB clean speech DCR (6 conditions per codec configuration):

-16 dBov clean channel - 9.6 kb/s, 13.2 kb/s, DTX OFF

-36 dBov clean channel - 24.4 kb/s, switching 32-128 kb/s, DTX ON

-26 dBov Profile 7(3.3%) - switching 9.6 - 24.4 kb/s, DTX ON

-26 dBov Profile 8(6.2%) - 13.2 kb/s CAM, DTX ON

Experiment 3 - SWB noisy speech DCR - 26 dBov, Street noise at 20 dB SNR (6 conditions per codec configuration):

clean channel - 9.6 kb/s, DTX ON

clean channel - 13.2 kb/s, DTX ON

clean channel - 24.4 kb/s, DTX ON

3% random FER - switching 9.6 - 24.4 kb/s, DTX ON

3% random FER - switching 32 - 128 kb/s, DTX ON

Profile 8(6.2%) - 13.2 kb/s CAM, DTX ON

Experiment 4 - SWB mixed and music DCR (6 conditions per codec configuration):

-16 dBov clean channel - 9.6 kb/s, DTX ON

-26 dBov clean channel - 13.2 kb/s, DTX ON

-36 dBov clean channel - 24.4 kb/s, DTX ON

-26 dBov 3% random FER - switching 9.6 - 24.4 kb/s, DTX ON

-26 dBov 3% random FER - switching 32 - 128 kb/s, DTX ON

-26 dBov Profile 8(6.2%) - 13.2 kb/s CAM, DTX ON

6 Conclusions

During the recent several years, processors with wide accumulators, SIMD support and VLIW features have become prevalent. On the other hand, the latest major update to the ITU-T Basic Operators [6] that serve as a foundation for reference software of codecs specified by 3GPP occurred in 2005, with a consequent update in 2009. EVS [2], the latest speech and audio codec standardized by 3GPP, was specified using those operators.

Given the information collected during the study, it is recommended to submit the proposed new set of basic operators to the STL GitHub open source environment as an extension of the current ITU-T Basic Operators. It is further recommended to inform ITU-T Study Group 12 of the new set of basic operators including the updated weights for the current set of basic operators, which have been agreed in 3GPP SA4, and request them to update Recommendation ITU-T G.191 (STL) [6] accordingly so that the new set of basic operators will be available for future codec standardizations.

It was further shown that by implementing the EVS codec using the new set of basic operators, a complexity gain of about 25% can be obtained for a given example of a modern processor. The corresponding decrease in WMOPS was about 10%. It is thus recommended to begin normative work with the objective of specifying an alternative implementation of the EVS codec using the new set of basic operators. The evaluation of the alternative implementation should follow the guidelines outlined in clause 5.3 of the present document.

Annex A: Extended Basic Operators

Name: enh64.c, enh32.c, complex_basop.c, enhUL32.c

Associated header file: enh64.h, enh32.h complex_basop.h, enhUL32.h

Variable definitions:

C_var1, C_var2: 16 bit complex variables

CL_var1, CL_var2: 32 bit complex variables

W_var1, W_var2: 64 bit variables

L_var1, L_var2: 32 bit variables

UL_var1, UL_var2, UL_varout_h, UL_varout_l: 32 bit unsigned variables

var1, var2: 16 bit variables

U_var1, U_varout_l: 16 bit unsigned variables

A.1 Basic operators that use 64 bit registers/accumulators

`W_add_nosat(W_var1, W_var2)`	Adds the two 64-bit variables W_var1 and W_var2 without saturation control on 64 bits.
`W_sub_nosat (W_var1, W_var2)`	Subtracts the two 64-bit variables W_var1 and W_var2 without saturation control on 64 bits.
`W_shl (W_var1, var2)`	Arithmetically shifts left the 64-bit variable W_var1 by var2 positions: if var2 is negative, W_var1 is shifted to the least significant bits by (–var2) positions with extension of the sign bit. if var2 is positive, W_var1 is shifted to the most significant bits by (var2) positions with saturation control on 64 bits.
W_shr (W_var1, var2)	Arithmetically shifts right the 64-bit variable W_var1 by var2 positions: if var2 is negative, W_var1 is shifted to the most significant bits by (–var2) positions with saturation control on 64 bits . if var2 is positive, W_var1 is shifted to the least significant bits by (var2) positions with extension of the sign bit.

W_shl_nosat (W_var1, var2)	Arithmetically shifts left the 64-bit variable W_var1 by var2 positions: if var2 is negative, W_var1 is shifted to the least significant bits by (–var2) positions with extension of the sign bit. if var2 is positive, W_var1 is shifted to the most significant bits by (var2) positions without saturation control on 64 bits.
W_shr_nosat (W_var1, var2)	Arithmetically shifts right the 64-bit variable W_var1 by var2 positions: if var2 is negative, W_var1 is shifted to the most significant bits by (–var2) positions without saturation control on 64 bits . if var2 is positive, W_var1 is shifted to the least significant bits by (var2) positions with extension of the sign bit.
W_mult_32_16 (L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 and sign extend to 64-bits without saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the result is produced in 17Q47 format.
W_mac_32_16 (W_acc, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 and sign extend to 64-bits without saturation control; add this 64 bit value to the 64 bit W_acc without saturation control, and return a 64 bit result The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then added to W_acc (in 17Q47) format. The final result is in 17Q47 format.
W_msu_32_16 (W_acc, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 and sign extend to 64-bits without saturation control; subtract this 64 bit value from the 64 bit W_acc without saturation control, and return a 64 bit result. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then subtracted from W_acc (in 17Q47) format. The final result is in 17Q47 format.
W_mult0_16_16 (var1, var2)	Multiply 16 bit var1 by 16 bit var2, sign extend to 64 bits and return the 64 bit result.

W_mac0_16_16 (W_acc, var1, var2)	Multiply 16 bit var1 by 16 bit var2, sign extend to 64 bits; add this 64 bit value to the 64 bit W_acc without saturation control, and return a 64 bit result
W_msu0_16_16 (W_acc, var1, var2)	Multiply 16 bit var1 by 16 bit var2, sign extend to 64 bits; subtract this 64 bit value from the 64 bit W_acc without saturation control, and return a 64 bit result.
W_mult_16_16 (W_acc, var1, var2)	Multiply a signed 16 bit var1 by signed 16 bit var2, shift the product left by 1 and sign extend to 64-bits without saturation control and return a 64 bit result The operation is performed in fractional mode. For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the result is produced in 33Q31 format.
W_mac_16_16 (W_acc, var1, var2)	Multiply a signed 16 bit var1 by signed 16 bit var2, shift the result left by 1 and sign extend to 64-bits; add this 64 bit value to the 64 bit W_acc without saturation control, and return a 64 bit result The operation is performed in fractional mode. For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the product is in 33Q31 format which is then added to W_acc (in 33Q31 format) to provide a final result in 33Q31 format.
W_msu_16_16 (W_acc, var1, var2)	Multiply a signed 16 bit var1 by signed 16 bit var2, shift the result left by 1 and sign extend to 64-bits; subtract this 64 bit value from the 64 bit W_acc without saturation control, and return a 64 bit result The operation is performed in fractional mode. For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the product is in 33Q31 format which is then subtracted from W_acc (in 33Q31 format) to provide a final result in 33Q31 format.
W_deposit32_l (L_var1)	Deposit the 32 bit L_var1 into the 32 LS bits of the 64 bit output. The 32 MS bits of the output are sign extended
W_deposit32_h (L_var1)	Deposit the 32 bit L_var1 into the 32 MS bits of the 64 bit output. The 32 LS bits of the output are zeroed.

W_sat_l (W_var)	Saturate the 64 bit variable W_var to 32 bit value and return the lower 32 bits. For example, a 64b wide accumulator is helpful in accumulating 16*16 multiplies without checking for saturation. However, at the end of the multiply-and-accumulate loop, we need to return only the 32b value after checking for saturation. If W_var is in 33Q31 format, then the result returned will be saturated to 1Q31 format.
W_sat_m (W_var)	Arithmetic right shift the 64 bit variable W_var by 16 bits; saturate the 64 bit value to 32 bit value and return the lower 32 bits. For example, a 64 bit wide accumulator is helpful in accumulating 32*16 multiplies without checking for saturation. A 32*16 multiply gives a 48 bit product; at the end of the multiply-and-accumulate loop, the result is in the lower 48 bits of the 64 bit accumulator. Now an arithmetic right shift by 16 bits will drop the LSB 16 bits. Now we should check for saturation and return the lower 32 bits. If W_var is in 17Q47 format, then the result returned will be saturated to 1Q31 format.
W_shl_sat_l (W_var, var1)	Arithmetic left shift the 64 bit W_var by var1 positions with lower 32 bit saturation and return the 32 LSB of 64 bit result. If var1 is negative, the result is shifted to right by (-var1) positions and sign extended. After shift operation, returns the 32 MSB of 64 bit result.
W_extract_l (W_var1)	Return the 32 LSB of a 64 bit variable W_var1.
W_extract_h (W_var1)	Return the 32 MSB of a 64 bit variable W_var1.
W_round48_L (W_var1)	Rounds the lower 16 bits of the 64-bit input number W_var1 into the most significant 32 bits with saturation. Shifts the resulting bits right by 16 and returns the 32-bit number: If W_var1 is in 17Q47 format, then the result returned will be rounded and saturated to 1Q31 format.
W_round32_s (W_var1)	Rounds the lower 32 bits of the 64-bit input number W_var1 into the most significant 16 bits with saturation. Shifts the resulting bits right by 32 and returns the 16-bit number: If W_var1 is in 17Q47 format, then the result returned will be rounded and saturated to 1Q15 format.

W_norm (W_var1)	Produces the number of left shifts needed to normalize the 64-bit variable W_var1. If W_var1 contains 0, return 0
W_add (W_var1, W_var2)	Adds the two 64-bit variables W_var1 and W_var2 with 64-bit saturation control. Sets overflow flag. Returns 64-bit result.
W_sub (W_var1, W_var2)	Subtracts 64-bit variable W_var2 from W_var1 with 64-bit saturation control. Sets overflow flag. Returns 64-bit result.
W_neg (W_var1)	Negates a 64-bit variables W_var1 with 64-bit saturation control. Set overflow flag. Returns 64-bit result.
W_abs (W_var1)	Returns a 64-bit absolute value of a 64-bit variable W_var1 with saturation control.
W_mult_32_32 (L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shift the product left by 1 with saturation control. Returns the 64-bit result. The operation is performed in fractional mode. For example, if L_var1 & L_var2 are in 1Q31 format then the result is produced in 1Q63 format. Note that W_mult_32_32(-2147483648, -2147483648) = 9223372036854775807.
W_mult0_32_32 (L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Returns the 64-bit result. For example, if L_var1 & L_var2 are in 1Q31 format then the result is produced in 2Q62 format.
W_lshl (W_var1, var2)	Logically shift the 64-bit input W_var1 left by var2 positions - If var2 is negative, logically shift right W_var1 by (-var2)
W_lshr (W_var1, var2)	Logically shift the 64-bit input W_var1 right by var2 positions - If var2 is negative, logically shift left W_var1 by (-var2)
W_round64_L (W_var1)	Rounds the lower 32 bits of the 64-bit input number W_var1 into the most significant 32 bits with saturation. Shifts the resulting bits right by 32 and returns the 32-bit number: If W_var1 is in 1Q63 format, then the result returned will be rounded and saturated to 1Q31 format.

A.2 Basic operators which use 32 bit precision multiply

Basic operators in this clause are useful for FFT and scaling functions where the result of a 32*16 or 32*32 arithmetic operation is rounded, and saturated to 32 bit value. There is no accumulation of products in these functions. In functions that accumulate products, you should use base operators in clause A.1.

Mpy_32_16_1(L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48 bit saturation control; Return the 32 MSB of the 48 bit result after truncation of lower 16 bits The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated and returned in 1Q31 format. Following code snippet describe the operations performed W_var1 = `W_mult_32_16 ( L_var1, var2 );` L_var_out = W_sat_m(W_var1);
Mpy_32_16_r(L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48 bit saturation control; Return the 32 MSB of the 48 bit result after rounding of the lower 16 bits The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then rounded, saturated, and returned in 1Q31 format. Following code snippet describe the operations performed W_var1 = `W_mult_32_16 ( L_var1, var2 );` L_var_out = W_round48_L (W_var1);

Mpy_32_32(L_var1, L_var2)

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shift the product left by 1 with 64 bit saturation control; Return the 32 MSB of the 64 bit result after truncating of the lower 32 bits

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q31 format, then the product is produced in 1Q63 format which is then truncated, saturated, and returned in 1Q31 format.

Following code snippet describe the operations performed

```
W_var1 = (( Word64)L_var1 * L_var2);

L_var_out = W_extract_h(W_shl(W_var1, 1) );
```

Mpy_32_32_r(L_var1, L_var2)

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Add rounding offset to lower 31 bits of the product. Shift the result left by 1 with 64 bit saturation control; return the 32 MSB of the 64 bit result with saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the result is produced in 1Q63 format which is then rounded, saturated, and returned in 1Q31 format.

Following code snippet describe the operations performed

```
W_var1 = (( Word64)L_var1 * L_var2);

W_var1 = W_var1 + 0x40000000LL;

W_var1 = W_shl ( W_var1, 1 );

L_var_out = W_extract_h( W_var1 );
```

Madd_32_16(L_var3, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48 bit saturation control; Add the 32 bit MSB of the 48 bit result with 32 bit L_var3 with 32 bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated to 1Q31 format and added to L_var3 in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_16_1(L_var1, var2); L_var_out = L_add(L_var3, L_var_out);
Madd_32_16_r(L_var3, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48-bit saturation control; Get the 32-bit MSB from 48-bit result after rounding of the lower 16 bits and add this with 32-bit L_var3 with 32-bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, rounded to 1Q31 format and added to L_var3 in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_16_r(L_var1, var2); L_var_out = L_add(L_var3, L_var_out);

Msub_32_16(L_var3, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48 bit saturation control; Subtract the 32 bit MSB of the 48 bit result from 32 bit L_var3 with 32 bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated to 1Q31 format and subtracted from L_var3 in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_16_1(L_var1, var2); L_var_out = L_sub(L_var3, L_var_out);
Msub_32_16_r(L_var3, L_var1, var2)	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48-bit saturation control; Get the 32-bit MSB from 48-bit result after rounding of the lower 16 bits and subtract this from 32-bit L_var3 with 32-bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, rounded to 1Q31 format and subtracted from L_var3 in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_16_r(L_var1, var2); L_var_out = L_sub(L_var3, L_var_out);

Madd_32_32(L_var3, L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shift the product left by 1 with 64 bit saturation control; Add the 32 MSB of the 64 bit result to 32 bit signed variable L_var3 with 32 bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and truncated in 1Q31 format which is then added to L_var3 (in 1Q31 format), to provide result in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_32(L_var1, L_var2); L_var_out = L_add(L_var3, L_var_out);
Madd_32_32_r(L_var3, L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Add rounding offset to lower 31 bits of the product. Shift the result left by 1 with 64-bit saturation control; get the 32 MSB of the 64-bit result with saturation and add this with 32-bit signed variable L_var3 with 32-bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and rounded in 1Q31 format which is then added to L_var3 (in 1Q31 format), to provide result in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_32_r(L_var1, L_var2); L_var_out = L_add(L_var3, L_var_out);

Msub_32_32(L_var3, L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shift the product left by 1 with 64 bit saturation control; Subtract the 32 MSB of the 64 bit result from 32 bit signed variable L_var3 with 32 bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and truncated in 1Q31 format which is then subtracted from L_var3 (in 1Q31 format), to provide result in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_32(L_var1, L_var2); L_var_out = L_sub(L_var3, L_var_out);
Msub_32_32_r(L_var3, L_var1, L_var2)	Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Add rounding offset to lower 31 bits of the product. Shift the result left by 1 with 64-bit saturation control; get the 32 MSB of the 64-bit result with saturation and Subtract this from 32-bit signed variable L_var3 with 32-bit saturation control. The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and rounded in 1Q31 format which is then subtracted from L_var3 (in 1Q31 format), to provide result in 1Q31 format. Following code snippet describe the operations performed L_var_out = Mpy_32_32_r(L_var1, L_var2); L_var_out = L_sub(L_var3, L_var_out);

A.3 Basic operators which use complex data types

CL_shr (CL_var1, var2)	Arithmetically shifts right the real and imaginary parts of the 32 bit complex number CL_var1 by var2 positions If var2 is negative, real and imaginary parts of CL_var1 are shifted to the most significant bits by (-var2) positions with 32-bit saturation control. If var2 is positive, real and imaginary parts of CL_var1 are shifted to the least significant bits by (var2) positions with sign extension Following code snippet describe the operations performed on real & imaginary part of a complex number. CL_result.re = L_shr(CL_var1.re, L_shift_val); CL_result.im = L_shr(CL_var1.im, L_shift_val);
CL_shl (CL_var1, var2)	Arithmetically shift left the real and imaginary parts of the 32 bit complex number CL_var1 by L_shift_val positions If var2 is negative, real and imaginary parts of CL_var1 are shifted to the least significant bits by (-var2) positions with sign extension If var2 is positive, real and imaginary parts of CL_var1 are shifted to the most significant bits by (var2) positions with 32-bit saturation control Following code snippet describe the operations performed on real & imaginary part of a complex number. CL_result.re = L_shl(CL_var1.re, L_shift_val); CL_result.im = L_shl(CL_var1.im, L_shift_val);

CL_add (CL_var1, CL_var2)	Adds the two 32 bit complex numbers CL_var1 and CL_var2 with 32-bit saturation control. Real part of the 32 bit complex number CL_var1 is added to Real part of the 32 bit complex number CL_var2 with 32 bit saturation control. The result forms the real part of the result variable. Imaginary part of the 32 bit complex number CL_var1 is added to Imaginary part of the 32 bit complex number CL_var2 with 32 bit saturation control. The result forms the imaginary part of the result variable. Following code snippet describe the operations performed on real & imaginary part of a complex number. CL_result.re = L_add(CL_var1.re, CL_var2.re); CL_result.im = L_add(CL_var1.im, CL_var2.im);
CL_sub (CL_var1, CL_var2)	Subtract the two 32 bit complex numbers CL_var1 and CL_var2 with 32-bit saturation control Real part of the 32 bit complex number CL_var2 is subtracted from Real part of the 32 bit complex number CL_var1 with 32 bit saturation control. The result forms the real part of the result variable. Imaginary part of the 32 bit complex number CL_var2 is subtracted from Imaginary part of the 32 bit complex number CL_var1 with 32 bit saturation control. The result forms the imaginary part of the result variable. Following code snippet describe the operations performed on real & imaginary part of a complex number. CL_result.re = L_sub(CL_var1.re, CL_var2.re); CL_result.im = L_sub(CL_var1.im, CL_var2.im);

CL_scale (CL_var, var1)	Multiply the real and imaginary parts of a 32 bit complex number CL_var by a 16-bit var1. The resulting 48 bit product for each part is rounded, saturated and 32 bit MSB of 48 bit result are returned.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = Mpy_32_16_r(CL_var.re, var1);
	CL_result.im = Mpy_32_16_r(CL_var.im, var1);

CL_dscale (CL_var, var1, var2)	Multiply the real parts of a 32 bit complex number CL_var by a 16-bit var1 and imaginary parts of a 32 bit complex number CL_var by a 16-bit var2. The resulting 48 bit product for each part is rounded, saturated and 32 bit MSB of 48 bit result are returned.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = Mpy_32_16_r(CL_var.re, var1);
	CL_result.im = Mpy_32_16_r(CL_var.im, var2);

CL_msu_j (CL_var1, CL_var2)	Multiply the 32 bit complex number CL_var2 with j and subtract the result from the 32 bit complex number CL_var1 with saturation control.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = L_add(CL_var1.re, CL_var2.im);
	CL_result.im = L_sub(CL_var1.im, CL_var2.re);

CL_mac_j (CL_var1, CL_var2)	Multiply the 32 bit complex number CL_var2 with j and add the result to the 32 bit complex number CL_var1 with saturation control.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = L_sub(CL_var1.re, CL_var2.im);
	CL_result.im = L_add(CL_var1.im, CL_var2.re);

CL_move (CL_var)	Copy the 32 bit complex number CL_var to destination 32 bit complex number
CL_Extract_real (CL_var)	Return the real part of a 32 bit complex number CL_var
CL_Extract_imag (CL_var)	Return the imaginary part of a 32 bit complex number CL_var

CL_form (L_re, L_im)	Combine the two 32 bit variable L_re and L_im and return a 32 bit complex number.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = L_re;
	CL_result.im = L_im;

CL_multr_32x16(CL_var, C_coeff)	Multiplication of 32 bit complex number CL_var with a 16 bit complex number C_coeff.
	
	The formula for multiplying two complex numbers, (x+iy) and (u+iv) is
	
	(x+iy)*(u+iv) = (xu – yv) + i(xv + yu);
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	W_tmp1 = W_mult_32_16(CL_var.re, C_coeff.re);
	
	W_tmp2 = W_mult_32_16(CL_var.im, C_coeff.im);
	
	W_tmp3 = W_mult_32_16(CL_var.re, C_coeff.im);
	
	W_tmp4 = W_mult_32_16(CL_var.im, C_coeff.re);
	
	CL_res.re = W_round48_L(W_sub_nosat (W_tmp1, W_tmp2));
	
	CL_res.im = W_round48_L(W_add_nosat (W_tmp3, W_tmp4));
	
	For example, if the real and imaginary part of complex variable CL_var are in 1Q31 format, and C_coeff in 1Q15 format, then the intermediate products would be in 17Q47 format. The round operation will convert the result of addition/subtraction from 17Q47 format to 1Q31 format.

CL_negate (CL_var)	Negate the 32 bit complex number, saturate and return.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = L_negate(CL_var.re);
	
	CL_result.im = L_negate(CL_var.im);

CL_conjugate(CL_var)	Negate only the imaginary part of complex number CL_var with saturation. No change in the real part.
	
	Following code snippet describe the operations
	
	CL_result.re = CL_var.re;
	
	CL_result.im = L_negate(CL_var.im);

CL_mul_j (CL_var)	Multiplication of a 32 bit complex number CL_var with j and return a 32 bit complex number.

CL_swap_real_imag (CL_var)	Swap real and imaginary parts of a 32 bit complex number CL_var and return a 32 bit complex number.

C_add (C_var1, C_var2)	Adds the two 16 bit complex numbers C_var1 and C_var2 with 16-bit saturation control.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	C_result.re = add(C_var1.re, C_var2.re);
	
	C_result.im = add(C_var1.im, C_var2.im);

C_sub (C_var1, C_var2)	Subtract the two 16 bit complex numbers C_var1 and C_var2 with 16-bit saturation control
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	C_result.re = sub(C_var1.re, C_var2.re);
	
	C_result.im = sub(C_var1.im, C_var2.im);

C_mul_j (C_var)	Multiplication of a 16 bit complex number with j and return a 16 bit complex number

C_multr (C_var1, C_var2)	Multiplication of 16 bit complex number C_var1 with 16 bit complex number C_var2 which results in a 16 bit complex number.
	The formula for multiplying two complex numbers, (x+iy) and (u+iv) is
	(x+iy)*(u+iv) = (xu – yv) + i(xv + yu);
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	W_tmp1 = W_mult_16_16(C_var1.re, C_var2.re);
	W_tmp2 = W_mult_16_16(C_var1.im, C_var2.im);
	W_tmp3 = W_mult_16_16(C_var1.re, C_var2.im);
	W_tmp4 = W_mult_16_16(C_var1.im, C_var2.re);
	C_result.re = round_fx(W_sat_l (W_sub_nosat (W_tmp1, W_tmp2)));
	C_result.im = round_fx(W_sat_l (W_add_nosat (W_tmp3, W_tmp4)));

C_form (re, im)	Combine the two 16 bit variable re and im and return a 16 bit complex number
CL_scale_32(CL_var1, L_var2)	Multiply the real and imaginary parts of a 32-bit complex number CL_var1 by a 32-bit L_var2. The resulting 64-bit product for each part is rounded, saturated and 32 bit MSB of 64-bit result are returned. Following code snippet describe the operations performed on real & imaginary part of a complex number. CL_result.re = Mpy_32_32_r(CL_var1.re, L_var2); CL_result.im = Mpy_32_32_r(CL_var1.im, L_var2);

CL_dscale_32(CL_var1, L_var2, L_var3)	Multiply the real parts of a 32-bit complex number CL_var1 by a 32-bit L_var2 and imaginary parts of a 32-bit complex number CL_var1 by a 32-bit L_var3. The resulting 64-bit product for each part is rounded, saturated and 32 bit MSB of 64-bit result are returned.
	
	Following code snippet describe the operations performed on real & imaginary part of a complex number.
	
	CL_result.re = Mpy_32_32_r(CL_var1.re, L_var2);
	
	CL_result.im = Mpy_32_32_r(CL_var1.im, L_var3);

CL_multr_32x32(CL_var1, CL_var2)	complex multiplication of CL_var1 and CL_var2. Multiplication is in fractional mode. Both input and outputs are in 1Q31 format.
	
	W_tmp1 = W_mult_32_32(CL_var1.re, CL_var2.re);
	W_tmp2 = W_mult_32_32(CL_var1.im, CL_var2.im);
	W_tmp3 = W_mult_32_32(CL_var1.re, CL_var2.im);
	W_tmp4 = W_mult_32_32(CL_var1.im, CL_var2.re);
	
	CL_res.re = W_round64_L(W_sub (W_tmp1, W_tmp2));
	CL_res.im = W_round64_L(W_add (W_tmp3, W_tmp4));

C_mac_r(CL_var1, C_var2, var3)	Multiplies real and imaginary part of C_var2 by var3 and shifts the result left by 1. Adds the 32-bit result to CL_var1 with saturation. Rounds the 16 least significant bits of the result into the 16 most significant bits with saturation and shifts the result right by 16. Returns a 16-bit complex result.
	
	C_result = CL_round32_16(CL_add(Cl_var1,
	C_scale(C_var2, var3)));

C_msu_r(CL_var1, C_var2, var3)	Multiplies real and imaginary part of C_var2 by var3 and shifts the result left by 1. Subtract the 32-bit result from CL_var1 with saturation. Rounds the 16 least significant bits of the result into the 16 most significant bits with saturation and shifts the result right by 16. Returns a 16-bit complex result.
	
	C_result = CL_round32_16(CL_sub(Cl_var1,
	C_scale(C_var2, var3)));

CL_round32_16(CL_var1)	Rounds the lower 16 bits of the 32-bit complex number CL_var1 into the most significant 16 bits with saturation. Shifts the resulting bits right by 16 and returns the 16-bit complex number. If real and imaginary of CL_var1 is in 1Q31 format, then the result returned will be rounded and saturated to 1Q15 format.
C_Extract_real(C_var1)	Return the real part of a 16-bit complex number C_var
C_Extract_imag (C_var1)	Return the imaginary part of a 16-bit complex number C_var

C_scale(C_var1,var2)	Multiply the real and imaginary parts of a 16-bit complex number C_var by a 16-bit var1. Returns 32-bit complex number
C_negate(C_var1)	Negate the 16-bit complex number, saturate and return a 16 bit complex number.
C_conjugate(C_var1)	Negate only the imaginary part of a 16 bit complex number C_var1 with saturation. No change in the real part.
C_shr(C_var1, var2)	Arithmetically shifts right the real and imaginary parts of the 16-bit complex number C_var1 by var2 positions. If var2 is negative, real and imaginary parts of C_var1 are shifted to the most significant bits by (-var2) positions with 16-bit saturation control. If var2 is positive, real and imaginary parts of C_var1 are shifted to the least significant bits by (var2) positions with sign extension
C_shl(C_var1,var2)	Arithmetically shift left the real and imaginary parts of the 16-bit complex number C_var1 by var2 positions If var2 is negative, real and imaginary parts of C_var1 are shifted to the least significant bits by (-var2) positions with sign extension If var2 is positive, real and imaginary parts of C_var1 are shifted to the most significant bits by (var2) positions with 16-bit saturation control

A.4 Basic operators for control operation

The following basic operators should be used in the control processing part of the reference code. They are expected to help compilers generate more efficient code for control sections of the reference C code. In addition, they also help in computing a more accurate representation of control code operations in the total WMOPs (weighted millions of operations) of the reference code.

LT_16(var1, var2)	Return 1 if 16 bit variable var1 is less than 16 bit variable var2, else return 0
GT_16(var1, var2)	Return 1 if 16 bit variable var1 is greater than 16 bit variable var2, else return 0
LE_16(var1, var2)	Return 1 if 16 bit variable var1 is less than or equal to 16 bit variable var2, else return 0.
GE_16(var1, var2)	Return 1 if 16 bit variable var1 is greater than or equal to 16 bit variable var2, else return 0
EQ_16(var1, var2)	Return 1 if 16 bit variable var1 is equal to 16 bit variable var2, else return 0
NE_16(var1, var2)	Return 1 if 16 bit variable var1 is not equal to 16 bit variable var2, else return 0

LT_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is less than 32 bit variable L_var2, else return 0
GT_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is greater than 32 bit variable L_var2, else return 0
LE_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is less than or equal to 32 bit variable L_var2, else return 0
GE_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is greater than or equal to 32 bit variable L_var2, else return 0.
EQ_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is equal to 32 bit variable L_var2, else return 0
NE_32(L_var1, L_var2)	Return 1 if 32 bit variable L_var1 is not equal to 32 bit variable L_var2, else return 0
LT_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is less than 64 bit variable W_var2, else return 0
GT_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is greater than 64 bit variable W_var2, else return 0
LE_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is less than or equal to 64 bit variable W_var2, else return 0
GE_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is greater than or equal to 64 bit variable W_var2, else return 0.
NE_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is not equal to 64 bit variable W_var2, else return 0
EQ_64(W_var1, W_var2)	Return 1 if 64 bit variable W_var1 is equal to 64 bit variable W_var2, else return 0

A.5 Basic operators for unsigned data types

UL_addNs (UL_var1, UL_var2, *var1)	Adds the two unsigned 32-bit variables UL_var1 and UL_var2 with overflow control, but without saturation. Returns 32-bit unsigned result. var1 is set to 1 if wrap around occurred, otherwise 0.

UL_subNs (UL_var1, UL_var2, *var1)	Subtracts the 32-bit usigned variable UL_var2 from the 32-bit unsigned variable UL_var1 with overflow control, but without saturation. Returns 32-bit unsigned result. var1 is set to 1 if wrap around (to "negative") occurred, otherwise 0.
norm_ul (UL_var1)	Produces the number of left shifts needed to normalize the 32-bit unsigned variable UL_var1 for positive values on the interval with minimum of 0 and maximum of 0xffffffff. If UL_var1 contains 0, return 0.
UL_Mpy_32_32(UL_var1, UL_var2)	Multiplies the two unsigned values UL_var1 and UL_var2 and returns the lower 32 bits, without saturation control. UL_var1 and UL_var2 are supposed to be in Q32 format. The result is produced in Q64 format, the 32 LS bits. Operates like a regular 32x32-bit unsigned int multiplication in ANSI-C.
Mpy_32_32_uu(UL_var1, UL_var2, *UL_var3, *UL_var4)	Multiplies the two unsigned 32-bit variables UL_var1 and UL_var2. The operation is performed in fractional mode. UL_var1 and UL_var2 are supposed to be in Q32 format. The result is produced in Q64 format: UL_varout_h points to the 32 MS bits while UL_varout_l points to the 32 LS bits.
Mpy_32_16_uu(UL_var1, U_var1, UL_varout_h, U_varout_l)	Multiplies the unsigned 32-bit variable UL_var1 with the unsigned 16-bit variable U_var1. The operation is performed in fractional mode : UL_var1 is supposed to be in Q32 format. U_var1 is supposed to be in Q16 format. The result is produced in Q48 format: UL_varout_h points to the 32 MS bits while U_varout_l points to the 16 LS bits.
UL_deposit_l(U_var1)	Deposit the 16-bit U_var1 into the 16 LS bits of the 32-bit output. The 16 MS bits of the output are not sign extended.

Annex B:
Weights of the STL basic operators

This annex contains a list of the existing STL2009 and new extensions referred to as STL2017 basic operators and their weights for the modern DSP architectures.

Legends	
⬜	STL-2009 basic operators
🟦	STL-2017 Complex basic operators
🟧	STL-2017 64-bit basic operators
🟩	STL-2017 Enhanced 32-bit basic operators
🟪	STL-2017 Unsigned basic operators
🟨	STL-2017 Control code basic operators

	Complexity Weights		
BASOPS	**Existing STL2009 as is**	**Updated**	**Comments**
add	1	1	
sub	1	1	
abs_s	1	1	
shl	1	1	
shr	1	1	
extract_h	1	1	
extract_l	1	1	
mult	1	1	
L_mult	1	1	
negate	1	1	
round	1	1	
L_mac	1	1	
L_msu	1	1	
L_macNs	1	1	
L_msuNs	1	1	
L_add	1	1	
L_sub	1	1	
L_add_c	2	2	
L_sub_c	2	2	
L_negate	1	1	
L_shl	1	1	
L_shr	1	1	
mult_r	1	1	
shr_r	3	2	Reduced to reflect modern processor architecture
mac_r	1	1	
msu_r	1	1	
L_deposit_h	1	1	
L_deposit_l	1	1	
L_shr_r	3	2	Reduced to reflect modern processor architecture
L_abs	1	1	
L_sat	4	1	Reduced to reflect modern processor architecture
norm_s	1	1	
div_s	18	18	
norm_l	1	1	
move16	1	1	
move32	2	1	Reduced to reflect modern processor architecture
Logic16	1	1	
Logic32	2	1	Reduced to reflect modern processor architecture
Test	2	1	Reduced to reflect modern processor architecture
s_max	1	1	
s_min	1	1	
L_max	1	1	

	Complexity Weights		
BASOPS	Existing STL2009 as is	Updated	Comments
L_min	1	1	
L40_max	1	1	
L40_min	1	1	
shl_r	3	2	Reduced to reflect modern processor architecture
L_shl_r	3	2	Reduced to reflect modern processor architecture
L40_shr_r	3	2	Reduced to reflect modern processor architecture
L40_shl_r	3	2	Reduced to reflect modern processor architecture
norm_L40	1	1	
L40_shl	1	1	
L40_shr	1	1	
L40_negate	1	1	
L40_add	1	1	
L40_sub	1	1	
L40_abs	1	1	
L40_mult	1	1	
L40_mac	1	1	
mac_r40	2	2	
L40_msu	1	1	
msu_r40	2	2	
Mpy_32_16_ss	2	2	Reduced to reflect modern processor architecture
Mpy_32_32_ss	4	2	Reduced to reflect modern processor architecture
L_mult0	1	1	
L_mac0	1	1	
L_msu0	1	1	
lshl	1	1	
lshr	1	1	
L_lshl	1	1	
L_lshr	1	1	
L40_lshl	1	1	
L40_lshr	1	1	
s_and	1	1	
s_or	1	1	
s_xor	1	1	
L_and	1	1	
L_or	1	1	
L_xor	1	1	
rotl	3	3	
rotr	3	3	
L_rotl	3	3	
L_rotr	3	3	
L40_set	3	1	Reduced to reflect modern processor architecture
L40_deposit_h	1	1	
L40_deposit_l	1	1	
L40_deposit32	1	1	
Extract40_H	1	1	
Extract40_L	1	1	
L_Extract40	1	1	
L40_round	1	1	
L_saturate40	1	1	
round40	1	1	
IF	4	3	
GOTO	4	2	
BREAK	4	2	
SWITCH	8	6	
FOR	3	3	
WHILE	4	3	
CONTINUE	4	2	
L_mls	5	1	Reduced to reflect modern processor architecture
div_l	32	32	
i_mult	3	1	Reduced to reflect modern processor architecture
CL_shr		1	
CL_shl		1	

BASOPS	Complexity Weights		Comments
	Existing STL2009 as is	Updated	
CL_add		1	
CL_sub		1	
CL_scale		1	
CL_dscale		1	
CL_msu_j		1	
CL_mac_j		1	
CL_move		1	
CL_Extract_real		1	
CL_Extract_imag		1	
CL_form		1	
CL_multr_32x16		2	
CL_negate		1	
CL_conjugate		1	
CL_mul_j		1	
CL_swap_real_imag		1	
C_add		1	
C_sub		1	
C_mul_j		1	
C_multr		2	
C_form		1	
CL_scale_32		1	
CL_dscale_32		1	
CL_multr_32x32		2	
C_mac_r		2	
C_msu_r		2	
CL_round32_16		1	
C_Extract_real		1	
C_Extract_imag		1	
C_scale		1	
C_negate		1	
C_conjugate		1	
C_shr		1	
C_shl		1	
move64		1	
W_add_nosat		1	
W_sub_nosat		1	
W_shl		1	
W_shr		1	
W_shl_nosat		1	
W_shr_nosat		1	
W_mac_32_16		1	SIMD and VLIW friendly basops
W_msu_32_16		1	SIMD and VLIW friendly basops
W_mult_32_16		1	SIMD and VLIW friendly basops
W_mult0_16_16		1	SIMD and VLIW friendly basops
W_mac0_16_16		1	SIMD and VLIW friendly basops
W_msu0_16_16		1	SIMD and VLIW friendly basops
W_mult_16_16		1	SIMD and VLIW friendly basops
W_mac_16_16		1	SIMD and VLIW friendly basops
W_msu_16_16		1	SIMD and VLIW friendly basops
W_shl_sat_l		1	
W_sat_l		1	
W_sat_m		1	
W_deposit32_l		1	
W_deposit32_h		1	
W_extract_l		1	
W_extract_h		1	
W_round48_L		1	
W_round32_s		1	
W_norm		1	
W_add		1	
W_sub		1	
W_neg		1	

BASOPS	Complexity Weights		Comments
	Existing STL2009 as is	Updated	
W_abs		1	
W_mult_32_32		1	
W_mult0_32_32		1	
W_lshl		1	
W_lshr		1	
W_round64_L		1	
Mpy_32_16_1		1	
Mpy_32_16_r		1	
Mpy_32_32		1	
Mpy_32_32_r		1	
Madd_32_16		1	
Madd_32_16_r		1	
Msub_32_16		1	
Msub_32_16_r		1	
Madd_32_32		1	
Madd_32_32_r		1	
Msub_32_32		1	
Msub_32_32_r		1	
LT_16		1	
GT_16		1	
LE_16		1	
GE_16		1	
EQ_16		1	
NE_16		1	
LT_32		1	
GT_32		1	
LE_32		1	
GE_32		1	
EQ_32		1	
NE_32		1	
LT_64		1	
GT_64		1	
LE_64		1	
GE_64		1	
EQ_64		1	
NE_64		1	

Annex C:
Change history

Change history							
Date	**Meeting**	**TDoc**	**CR**	**Rev**	**Cat**	**Subject/Comment**	**New version**
2017-12	SA#78	SP-170834				Presented to TSG SA#78 for information	1.0.0
2018-03	SA#79	SP-180029				Presented to TSG SA#79 for approval	2.0.0
2018-03						Version 15.0.0 approved at TSG SA#79	15.0.0
2018-09	SA#81	SP-180653	0004	2	F	Corrections, modification of Experiment 4 and addition to fixed-point basic operators	15.1.0

History

Document history		
V15.0.0	July 2018	Publication
V15.1.0	October 2018	Publication