Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«**СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**»
Институт космических и информационных технологий
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой

_____О.В.Непомнящий

«_____»_____ 20___г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 «Информатика и вычислительная техника»

Разработка и исследование голосового кодека для возможности применения в спутниковой связи

Руководитель _____ _____ проф., д-р техн. наук С. А. Бронов
подпись *дата* *должность, ученая степень*

Выпускник _____ _____ П. В. Винокуров
подпись *дата*

Нормоконтролер _____ _____ проф., д-р техн. наук С. А. Бронов
подпись *дата* *должность, ученая степень*

Красноярск 2022

Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«**СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**»
Институт космических и информационных технологий
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой

_____О. В. Непомнящий

«____»_____ 20___г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

в форме бакалаврской работы

Студенту Винокурову Павлу Владимировичу
Фамилия, имя, отчество

Группа КИ18-07Б **Направление** 09.03.01
номер *код*

Информатика и вычислительная техника
наименование

Тема выпускной квалификационной работы

Разработка и исследование голосового кодека для возможности применения в спутниковой связи

Утверждена приказом по университету № _____ от _____

Руководитель ВКР С. А. Бронов, канд. д-р техн. наук, профессор, руководи-
инициалы, фамилия, должность, учёное звание, место работы

тель НУЛ САПР каф. ВТ ИКИТ СФУ

Исходные данные для ВКР

Методические указания руководителя ВКР, публикации по теме работы.

Перечень разделов ВКР

Анализ предметной области, постановка задачи исследования, выбор методов исследования и инструментальных средств, разработка алгоритмов и их программная реализация, тестирование разработанной системы.

Перечень графического материала

Слайды презентации с математическими моделями, схемами алгоритмов и результатами работы разработанной системы.

Руководитель ВКР _____ С. А. Бронов
подпись

Задание принял к исполнению _____ П. В. Винокуров
подпись

«____» _____ 20____ г.

РЕФЕРАТ

Выпускная квалификационная работа на тему «Разработка и исследование голосового кодека для возможности применения в спутниковой связи» выполнен в научно-учебной лаборатории систем автоматизированного проектирования кафедры вычислительной техники института космических и информационных технологий Сибирского федерального университета.

АДИКМ, КОДЕК, СЖАТИЕ ЗВУКА

Объект исследования — процесс изучения проприетарного алгоритма кодирования звука ADPCM XAS.

Предмет исследования — алгоритм сжатия (кодирования) звука ADPCM EA XAS.

Объект разработки — кодировщик ADPCM EA XAS.

Цель работы — Анализ имеющегося алгоритма кодирования ADPCM EA XAS, написание кодировщика для него, анализ рациональности применения этого кодека для использования в системе спутниковой голосовой связи.

Результаты выполненной работы предполагается использовать для модификации файлов игр, использующих этот аудиокодек, а также возможно в системах спутниковой связи.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность. Одной стороны в среде энтузиастов есть интерес к модификации звуковых файлов для игр от разработчика Electronic Arts, использовавших проприетарную звуковую библиотеку с проприетарными форматами и проприетарными кодеками. Форматы были изучены достаточно, чтобы осуществить замену звуковых данных, но для кодеков были написаны только декодеры, что не позволяет вставить свой звук в файл игры.

С другой стороны, в системах спутниковой связи есть потребность в уплотнении канала связи, с использованием сжатия, в связи с чем была поставлена задача изучить рациональность применения данного аудио кодека в этой области.

Объект исследования — процесс изучения проприетарного алгоритма кодирования звука ADPCM EA XAS.

Предмет исследования — алгоритм сжатия (кодирования) звука ADPCM EA XAS.

Объект разработки — кодер/декодер ADPCM EA XAS.

Цель работы — Анализ имеющегося алгоритма кодирования ADPCM EA XAS, написание кодировщика для него, анализ рациональности применения этого кодека для использования в системе спутниковой голосовой связи.

Задачи работы:

1) Анализ имеющегося алгоритма декодирования ADPCM EA XAS и написание своего.

2) Анализ имеющегося в бинарном виде кодировщика для схожего алгоритма;

3) Написание кодировщика ADPCM XAS;

4) Изучение свойств получившегося кодека, с целью возможного применения в системах спутниковой связи.

Методы, инструментальные средства и технологии разработки. В качестве инструментального программного обеспечения использованы следующие программы:

- IDA (Interactive Disassembler) [2] — дизассемблер/отладчик для изучения программ в бинарном виде.

- Microsoft Visual Studio - Интерактивная среда разработки на языке C++.

Значение для практики заключается в том, что данный кодер/декодер можно будет использовать для работы с файлами игры и возможно в системах спутниковой связи.

1 Анализ задания на выпускную квалификационную работу

1.1 Имеющиеся средства работы с кодеком

Для алгоритма ADPCM EA XAS существуют следующие декодеры: ffmpeg с открытым исходным кодом и различные проприетарные программы от разработчика кодека, где он применяется. ADPCM EA XAS не имеет кодировщиков, его предстоит разработать. В файлах игр имеются образцы закодированного звука.

ADPCM XAS имеет предшественника — кодек ADPCM XA-R2, декодер для которого также есть в ffmpeg[3], а также есть проприетарный кодер от разработчика кодека.

Название кодека	Образцы закодированного звука (от разработчика)	Декодер с открытым исходным кодом	Проприетарный кодер
ADPCM XA-R2	+	+	+
ADPCM XAS	+	+	-

1.2 Разработка технического задания на кодер/декодер ADPCM XAS

Кодер/декодер должен представлять из себя библиотеку на языке C++ реализующую следующий интерфейс:

-Расчёт размера закодированной информации –

uint32_t GetXASEncodedSize(uint32_t n_samples_per_channel, uint32_t n_channels)

- Кодирование звука -

void encode_XAS(void* out_XAS, const int16_t* in_PCM, uint32_t n_samples_per_channel, uint32_t n_channels);

Исходный формат — ИКМ (PCM) 16 бит целые со знаком, в случае многоканального звука выборки из каждого канала смешиваются по очереди.

Выходной формат — Последовательность закодированных в XAS блоков (chunk) по 76 байт каждый, в случае многоканального звука блоки для каждого канала чередуются.

- Декодирование звука -

void decode_XAS(const void* in_XAS, int16_t* out_PCM, uint32_t n_samples_per_channel, uint32_t n_channels);

Исходный и выходной форматы идентичный случаю кодирования, но наоборот.

Кодек должен удовлетворять следующим требованиям:

- При декодировании и кодировании вновь закодированные данные должны быть идентичны исходным.

2 Написание декодера

Код из ffmpeg:

```
static const int16_t ea_adpcm_table[] = {
	0,  240,  460,  392,
	0,    0, -208, -220,
	0,    1,    3,    4,
	7,    8,   10,   11,
	0,   -1,   -3,   -4
};
...
CASE(ADPCM_EA_XAS,
	for (int channel = 0; channel < channels; channel++) {
		int coeff[2][4], shift[4];
		int16_t *s = samples_p[channel];
		for (int n = 0; n < 4; n++, s += 32) {
			int val = sign_extend(bytestream2_get_le16u(&gb), 16);
			for (int i = 0; i < 2; i++)
				coeff[i][n] = ea_adpcm_table[(val & 0x0F) + 4 * i];
			s[0] = val & ~0x0F;

			val = sign_extend(bytestream2_get_le16u(&gb), 16);
			shift[n] = 20 - (val & 0x0F);
			s[1] = val & ~0x0F;
		}

		for (int m = 2; m < 32; m += 2) {
			s = &samples_p[channel][m];
			for (int n = 0; n < 4; n++, s += 32) {
				int level, pred;
				int byte = bytestream2_get_byteu(&gb);

				level = sign_extend(byte >> 4, 4) * (1 << shift[n]);
				pred = s[-1] * coeff[0][n] + s[-2] * coeff[1][n];
				s[0] = av_clip_int16((level + pred + 0x80) >> 8);

				level = sign_extend(byte, 4) * (1 << shift[n]);
				pred = s[0] * coeff[0][n] + s[-1] * coeff[1][n];
				s[1] = av_clip_int16((level + pred + 0x80) >> 8);
			}
		}
	}
) /* End of CASE */
```

Похоже, что код был написан путем поверхностного анализа дизассемблированного кода какого-то исполняемого файла. Из этого кода трудно понять принцип работы кодека XAS, этот код нужно проанализировать и переписать.

Из названия кодека понятно, что он использует принцип Адаптивной Дифференциальной Импульсно-Кодовой Модуляции [4, 5] (АДИКМ, ADPCM)

2.1 Структура данных

Сперва нужно установить структуру блоков сжатых данных. Проанализировав код выше, можно определить следующую структуру: Каждый блок (chunk) состоит из четырех подблоков (subchunk), каждый подблок состоит из заголовка (XAS_SubChunkHeader - 4 байта) и 15 байт данных (XAS_data). Итого каждый блок занимает 76 байт. Заголовки каждого подблока чередуются, байты данных каждого подблока чередуются, как видно из дизассемблированного кода — это сделано для оптимизаций под векторные SIMD инструкции MMX и SSE.

```
punpcklwd mm1, mm0
punpckhwd mm2, mm3
movq    mm3, mm1
movq    mm0, mm2
pslld   mm0, 4
pslld   mm1, 4
pand    mm2, qword ptr [ebp-30h]
pand    mm3, qword ptr [ebp-30h]
cvtpi2ps xmm0, mm2
cvtpi2ps xmm1, mm0
movlhps xmm0, xmm0
movlhps xmm1, xmm1
cvtpi2ps xmm0, mm3
cvtpi2ps xmm1, mm1
movaps  xmm5, xmm6
mulps   xmm0, xmmword ptr [ebp-50h]
mulps   xmm5, xmmword ptr [ebp-60h]
mulps   xmm7, xmmword ptr [ebp-40h]
```

Рисунок 1 – Фрагмент дизассемблированного кода игры.

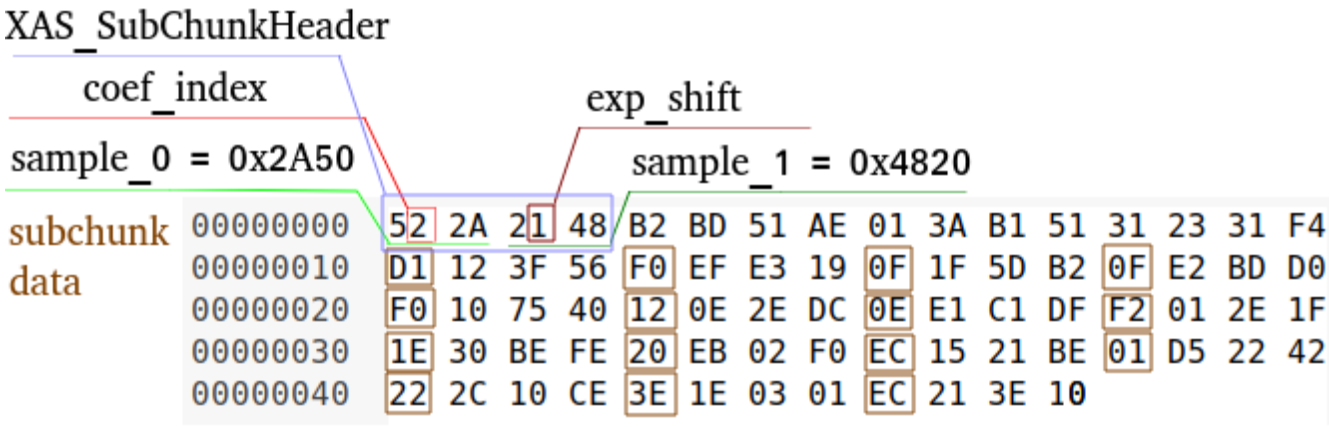
Опишем это следующим образом:

```
const int subchunks_in_XAS_chunk = 4;
struct XAS_Chunk {
    XAS_SubChunkHeader headers[subchunks_in_XAS_chunk];
    byte XAS_data[15][subchunks_in_XAS_chunk];
};
```

Заголовок подблока описывает первую выборку (sample_0 , 12 бит), модель предсказания (coef_index, 4 бита), вторую выборку (sample_1, 12 бит), экспоненту (exp_shift, 4 бита).

```
struct XAS_SubChunkHeader {
	unsigned coef_index : 2;
	unsigned unused : 2;
	signed sample_0 : 12;
	unsigned exp_shift : 4;
	signed sample_1 : 12;
	};
```

Каждый байт данных подблока содержит 2 по 4 бита коррекции ошибки



предсказания.

Рисунок 2 — Структура XAS блока

2.1 Алгоритм декодирования

1) Для каждого подблока берется 2 выборки, описанные в заголовке, присваиваем их в пред предыдущую (prev_prev_sample) и предыдущую (prev_sample) соответственно и расширенные до 16 бит со знаком, сохраняем их в поток как декодированные.

2) Выбираем 2 коэффициента (0 и 1) из таблицы (ea_adpcm_table) по номеру модели предсказания.

3) Коэффициент 1 умножаем на пред предыдущую выборку, коэффициент 0 умножаем на предыдущую выборку, складываем два результата, получаем предсказанную выборку (prediction).

4) Берем следующие 4 бита из данных подблока, интерпретируем как целое со знаком, умножаем на 2^(12 — exp_shift), получаем коррекцию ошибки (correction), эту операцию сделаем через битовый сдвиг.

5) Складываем предсказанную выборку с коррекцией ошибки,

6) Округляем до целого, усекаем полученное число до 16 бит со знаком, это будет декодированная выборка.

7) Сохраняем декодированную выборку в поток, пред предыдущей выборке присваиваем предыдущую, предыдущей присваиваем декодированную.

8) Повторяем шаги 3-7 для каждых 4-х бит из данных подблока (ещё 29 раз).

9) Повторяем шаги выше для каждого подблока.

10) Повторяем шаги выше для каждого блока.

Таблицу ea_adpcm_table возьмем из дизассемблированного кода NFSCarbon-v1.4.

```
.rdata:00A04330 ea_adpcm_table  dd 0.0                  ; DATA XREF: decode_EA_XAS+8B↑r
.rdata:00A04334 ; float flt_A04334[]
.rdata:00A04334 flt_A04334      dd 0.0                  ; DATA XREF: decode_EA_XAS+98↑r
.rdata:00A04338                 dd 0.9375
.rdata:00A0433C                 dd 0.0
.rdata:00A04340                 dd 1.796875
.rdata:00A04344                 dd -0.8125
.rdata:00A04348                 dd 1.53125
.rdata:00A0434C                 dd -0.859375
```

Рисунок 3 – Таблица из дизассемблированного кода

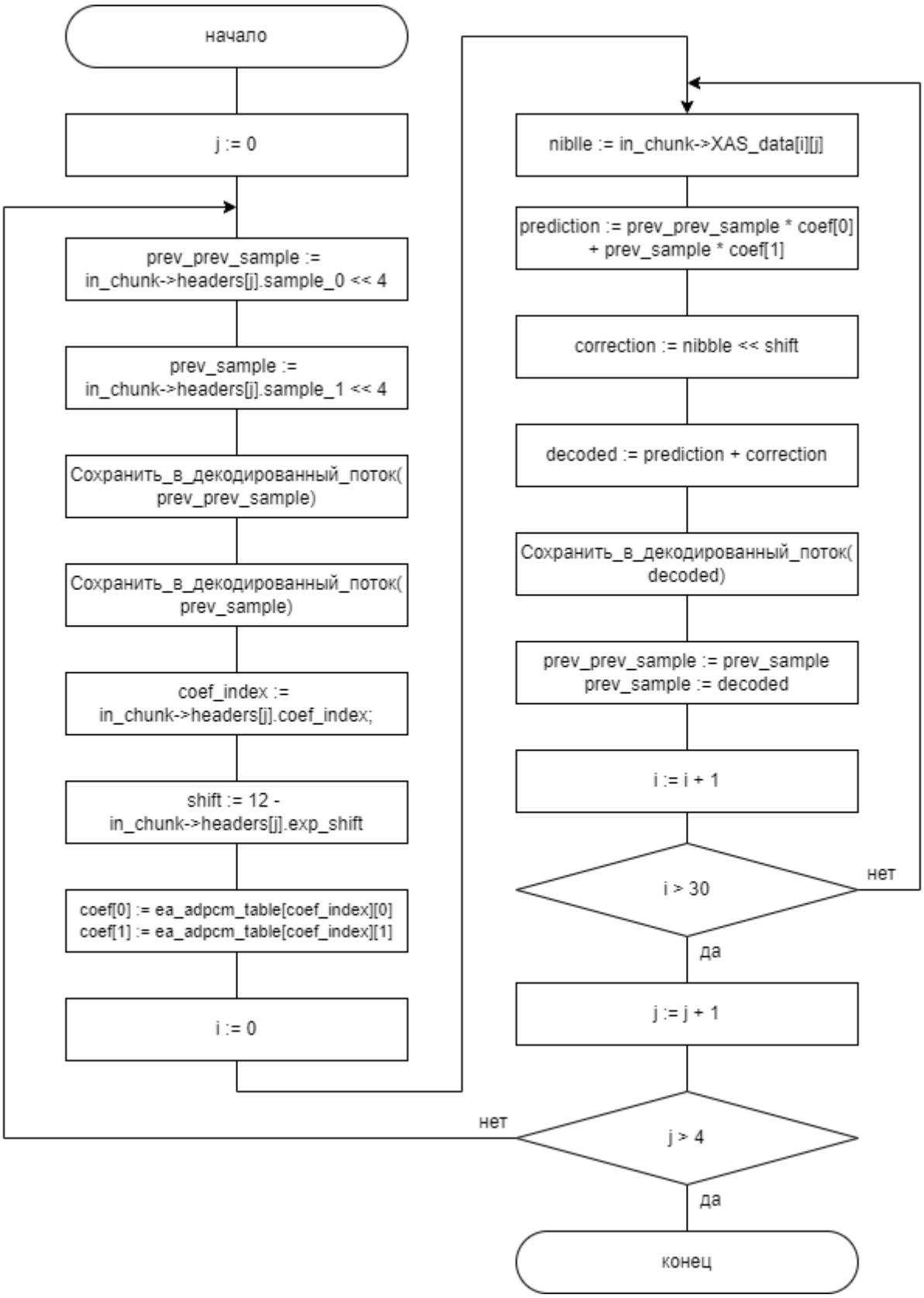


Рисунок 4 — Блок-схема алгоритма декодирования.

Продемонстрируем работу алгоритма на примере с подблоком, представленным на рисунке 2.

1) prev_prev_sample= $2A50_{16} = 10832_{10}$

prev_sample = $4820_{16} = 18464_{10}$

2) coef[2] = {1.796875, -0.8125}

shift = 12 - 1 = 11

3) prediction = 18464*1.796875 +10832*(-0.8125) = 24376,5

4) correction = $-3 * 2^{11} = -6144$

5, 6) decoded = Clip_int16(4461,75 + (-6144)) = Clip_int16(18232,5) = 18233

Внесем некоторые улучшения в алгоритм:

1) Чтобы алгоритм мог эффективно работать на процессорах не имеющих модуля вычислений с плавающей точкой, домножим коэффициенты из таблицы ea_adpcm_table на $2^{\text{fixed\_point\_offset}}$, где fixed_point_offset некоторое целое число, например 8, и преобразуем их в целые, а перед выполнением шага 6 поделим результат на $2^{\text{fixed\_point\_offset}}$ (реализуем через битовый сдвиг), для округления до целого перед этим прибавим $2^{\text{fixed\_point\_offset-1}}$, таким образом мы переходим от вычислений с плавающей точкой к вычислениям с фиксированной точкой. Выполнив домножение коэффициентов на 2^8, можно заметить, что числа из таблицы стали соответствовать числам из таблицы из кода ffmpeg.

2) Вынесем шаги 3-6 в отдельную функцию.

3) Предыдущие декодированные выборки будем брать из потока, а не из переменных.

Получаем следующий код для декодирования одного блока на языке C++:

```
const int def_bias_compens = (fixp_exponent >> 1);
inline int16_t decode_XA_sample(const int16_t prev_samples[2], const int coef[2], char
int4, byte shift, int bias_compens = def_bias_compens) {
	int correction = (int)int4 << shift;
	int prediction = prev_samples[1] * coef[0] + prev_samples[0] * coef[1] + bias_com-
pens;
	return Clip_int16((prediction + correction) >> fixed_point_offset);
}

void decode_XAS_Chunk(const XAS_Chunk* in_chunk, int16_t* out_PCM) {
	for (int j = 0; j < subchunks_in_XAS_chunk; j++) {
		int16_t *pSamples = out_PCM + j * 32;

		pSamples[0] = (in_chunk->headers[j].sample_0 << 4) + shift4_compens_bias;
		int coef_index = in_chunk->headers[j].coef_index;
```

```
                pSamples[1] = (in_chunk->headers[j].sample_1 << 4) + shift4_compens_bias;
                byte shift = 12 + fixed_point_offset - in_chunk->headers[j].exp_shift;

                const int* coef = ea_adpcm_table_v2[coef_index];

                for (int i = 0; i < 15; i++, pSamples += 2) {
                        SamplesByte data = *(SamplesByte*)&(in_chunk->XAS_data[i][j]);

                        pSamples[2] = decode_XA_sample(pSamples, coef, data.sample0, shift);
                        pSamples[3] = decode_XA_sample(pSamples + 1, coef, data.sample1,
shift);

                }
        }
        }
```

Посмотрим, насколько точны предсказания и сравним простым использованием предыдущей выборки вместо предсказания т.е. если бы использовалось предсказание с коэффициентами {1, 0}.

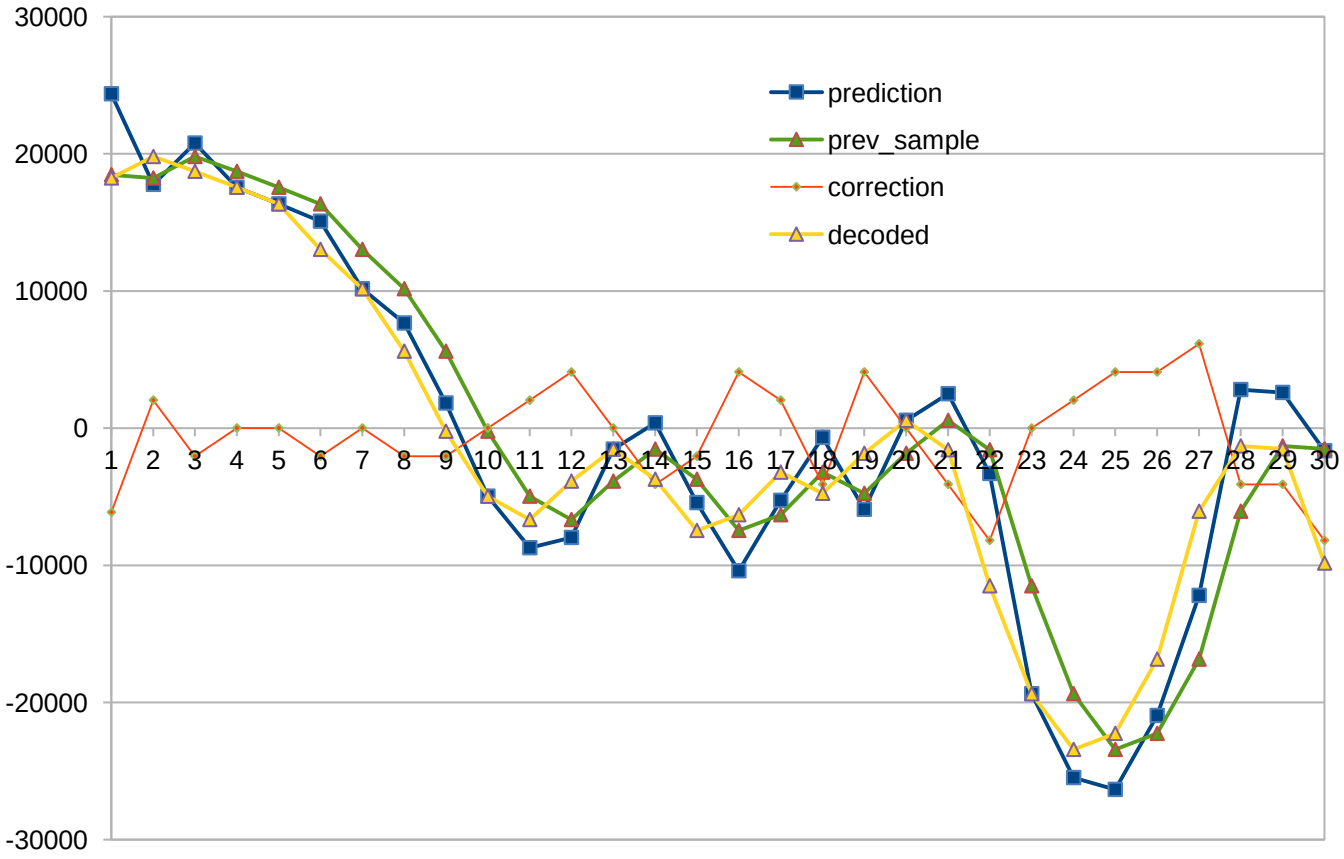


Рисунок 5 – Сравнение предсказаний

Остается тривиальная задача – написать функцию decode_XAS, где остается только правильно смешивать каналы.

3 Написание кодировщика

Для того, чтобы кодировать звук, нужно написать проделать шаги, обратные декодированию, однако возникает трудность: как выбрать coef_index и exp_shift?

Чтобы выбрать наилучшую модель предсказания, переберем все 4, и узнаем, какая дает наименьшую максимальную ошибку в подблоке.

```
int min_max_error = INT_MAX;
int s_min_max_error = INT_MAX;
int best_coef_ind = 0;
for (int coef_ind = 0; coef_ind < num_coefs; coef_ind++) {
	int16_t prevSamples[2] = { in_prevSamples[0], in_prevSamples[1] };
	int max_error = 0;
	int s_max_error = 0;
	for (int i = 0; i < num_samples; i++) {
		int prediction = ea_adpcm_table_v2[coef_ind][0] * prevSamples[1] +
ea_adpcm_table_v2[coef_ind][1] * prevSamples[0];
		int sample = pSamples[i];
		sample <<= fixed_point_offset;
		int s_error = sample - prediction;
		int error = abs(s_error);
		if (error > max_error) {
			max_error = error;
			s_max_error = s_error;
		}
		prevSamples[0] = prevSamples[1];
		prevSamples[1] = pSamples[i];
	}
	if (max_error < min_max_error) {
		min_max_error = max_error;
		best_coef_ind = coef_ind;
		s_min_max_error = s_max_error;
	}
}
```

Стоит заметить, что для вычисления предсказания используются оригинальные выборки, которые будут отличаться от декодированных, следовательно, значения ошибок будут отличаться.

Выбираем экспоненту для максимальной ошибки в подблоке:

```
int max_min_error_i16 = Clip_int16(min_max_error >> fixed_point_offset);
int mask = 0x4000;
int exp_shift;
for (exp_shift = 0; exp_shift < 12; exp_shift++) {
	if ((((mask >> 3) + max_min_error_i16) & mask) != 0) {
		break;
	}
	mask >>= 1;
}
```

После нахождения coef_index и exp_shift, можно делать тоже, что и при декодировании, но в обратном порядке:

```
int prediction = pDecodedSamples[1] * coef[0] + pDecodedSamples[0] * coef[1];

int correction = (sample << fixed_point_offset) + def_bias_compens - prediction;
int res = Clip_int4(correction >> shift);
```

Стоит помнить, что при декодировании предсказания будут вычисляться из декодированных ранее выборок, потому стоит иметь отельный поток для декодированных обратно выборок, а первые 2 выборки усечь до 12 бит.

```
int predecoded = ((res << shift) + prediction + def_bias_compens) >> fixed_point_offset;
int decoded = Clip_int16(predecoded);
```

Однако можно уменьшить потери, если вспомнить что выборки при декодировании усекаются до 16-и бит, то можно намеренно выходить за максимальное значение 32767 или минимальное -32768, пример показан на рисунке 6, очевидно, что значение INT16_MAX ближе к оригинальной выборке, чем значение, вычисленное при res = 3, поэтому лучше взять значение больше.

Рисунок 6 – Пример уменьшения потери

Учтём это в коде:

```
int term = 1 << (shift - fixed_point_offset);
int decoded2;
if (res != 7 && abs(decoded - sample) > abs((decoded2 = Clip_int16(predecoded + term)) -
sample)) {
	res += 1;
	decoded = decoded2;
}
else if (res != -8 && abs(decoded - sample) > abs((decoded2 = Clip_int16(predecoded -
term)) - sample)) {
	res -= 1;
	decoded = decoded2;
}
```

Чтобы не считать декодированных выборки заново для res + 1 и res - 1, добавим переменную term – единицу, сдвинутую на экспоненту.

Чтобы не переполнить 4-х битную переменную, проверяем, не достигла ли она уже максимального значения.

Анализ закодированных образцов звука показывает, что разработчик кодека тоже это учитывал.

Повторяем алгоритм для каждой выборки, каждого подблока и каждого блока.

4 Изучение свойств полученного кодека, с целью возможного применения в системах спутниковой связи

4.1 Особенности спутниковой связи

Наиболее распространенные спутники связи – спутники на геостационарной орбите.

Геостационарная орбита — это круговая орбита над экватором Земли, угловая скорость вращения на которой равна угловой скорости вращения Земли. При этом получается, что спутник почти неподвижен, относительно Земли [6].

Радиус геостационарной орбиты — 42 164 км, высота — 35 786 км.

Здесь и далее под спутниковой связью будет подразумеваться связь через спутник на геостационарной орбите.

Разделив высоту орбиты на скорость света получим задержку передачи сигнала примерно в 0.11937 с. Однако, 35 786 км — это только расстояние от спутника до точки на экваторе, над которой он висит, и чем дальше приемник/передатчик находится от спутника, тем больше будет расстояние и больше задержка сигнала. Например — ближайшее расстояние от Красноярска до геостационарной орбиты составляет 38 954 км, время прохождения сигнала 0,1299 с.

Спутник является не конечным получателем сигнала, а ретранслятором его на Землю, поэтому передача сигнала от одной наземной станции до другой через спутник займет 0.24 с, а минимальное время получения ответа около 0.48 с.

Еще одна особенность спутниковой связи — высокий уровень атмосферных помех, причем чем дальше от точки стояния спутника находится станция, тем больше атмосферы нужно пройти сигналу и тем большее будет помех. На уровень помех также влияют погодные явления. Это одна из причин, по которой в спутниковой связи используется цифровой сигнал, а для уменьшения ошибки используется помехозащищенное кодирование.

И наконец количество каналов на спутнике ограничено, поэтому рационально через один радиоканал передавать данные для нескольких абонентов с разделением по времени, при этом разные абоненты используют разные временные слоты (интервалы) для передачи.

4.2 Исследование потерь при кодировании

В данном исследовании участвовал образец звукового эффекта с разной частотой дискретизации.

Возьмем звуковой эффект с частотой дискретизации 44100 Гц.

Ощутимы на слух искажений после кодирования не замечено.

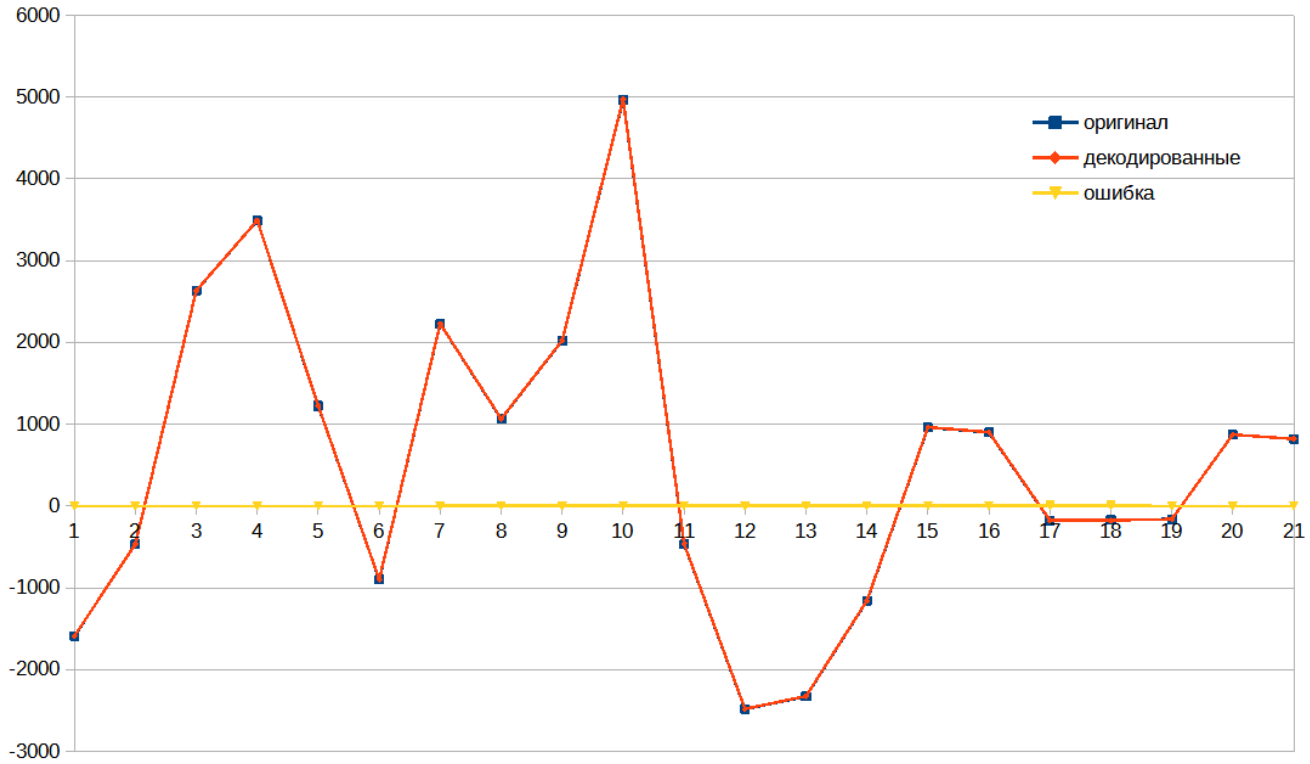


Рисунок 7 – Случайное место в звуковом эффекте

Наибольшая абсолютная ошибка — 2048 (22,45%)

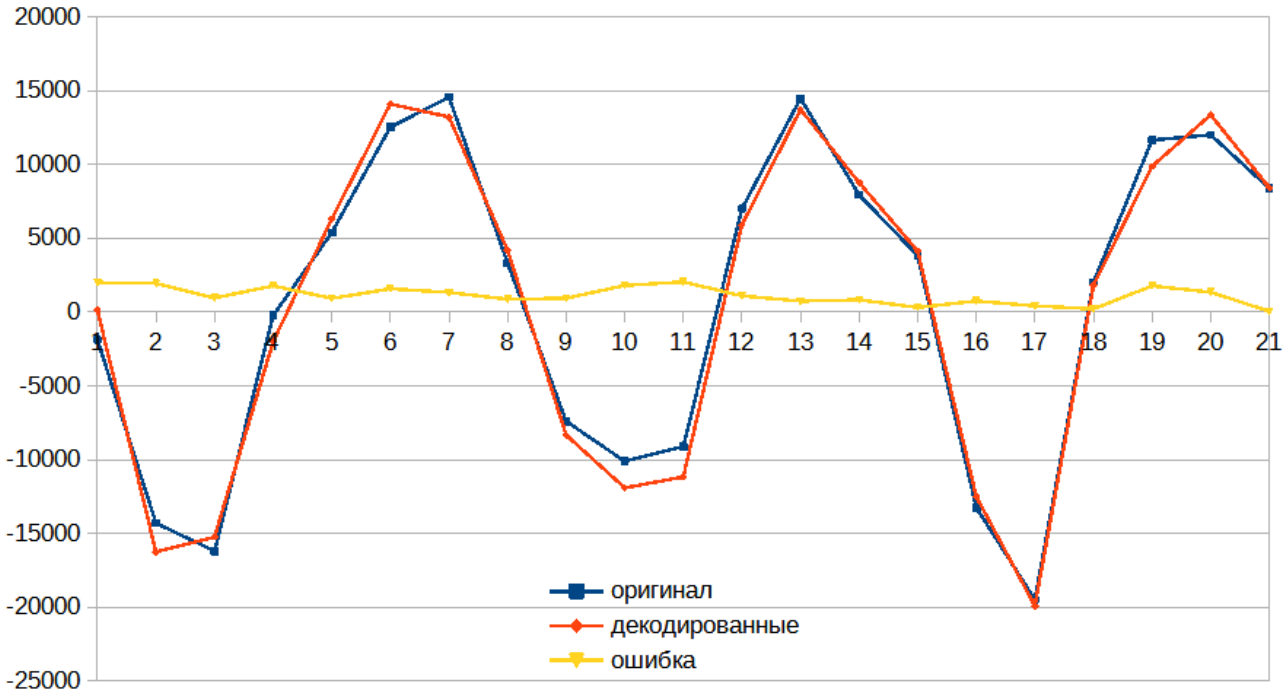


Рисунок 8 – Окрестности выборки с наибольшей абсолютной ошибкой

Среднеквадратичное отклонение — 357,14

Передескретизируем этот звук на 22020 Гц.

Качество звука несколько упало, в сравнении с 44100 Гц.

Искажения после кодирования/декодирования нет ощутимых искажений звука.

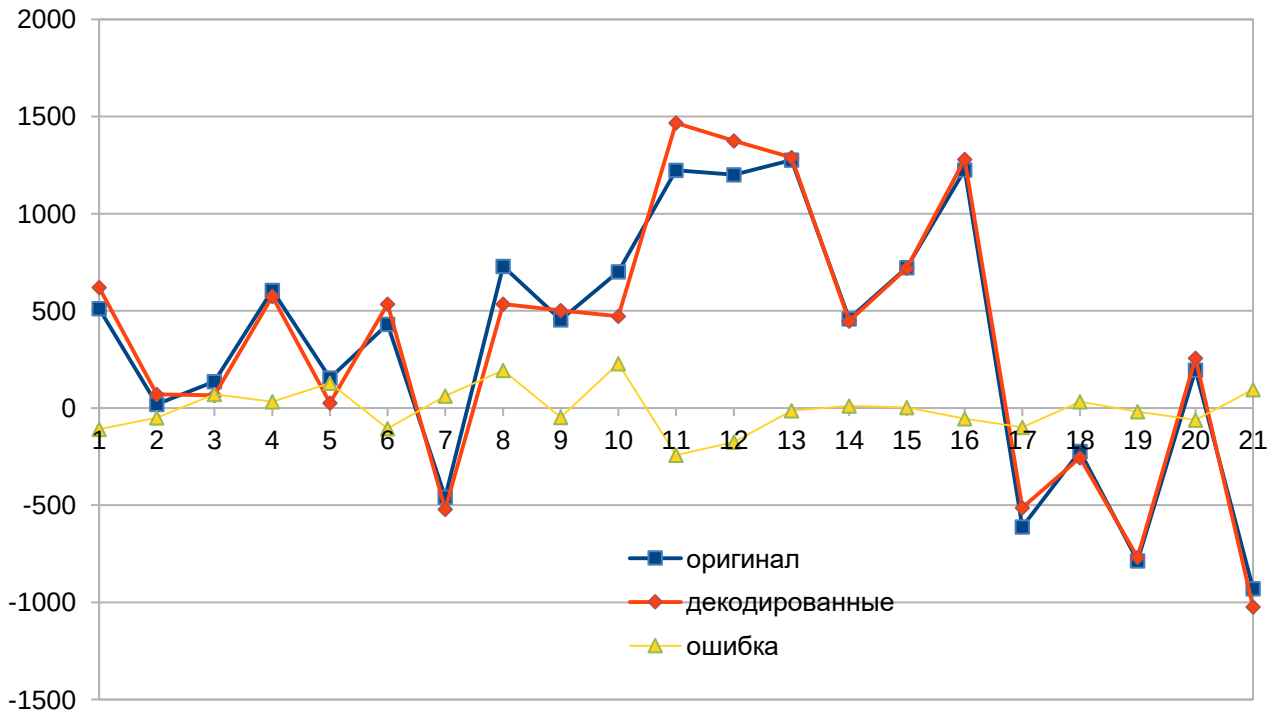


Рисунок 9 — Случайное место.

Наибольшая абсолютная ошибка — 2048 (33.3%)

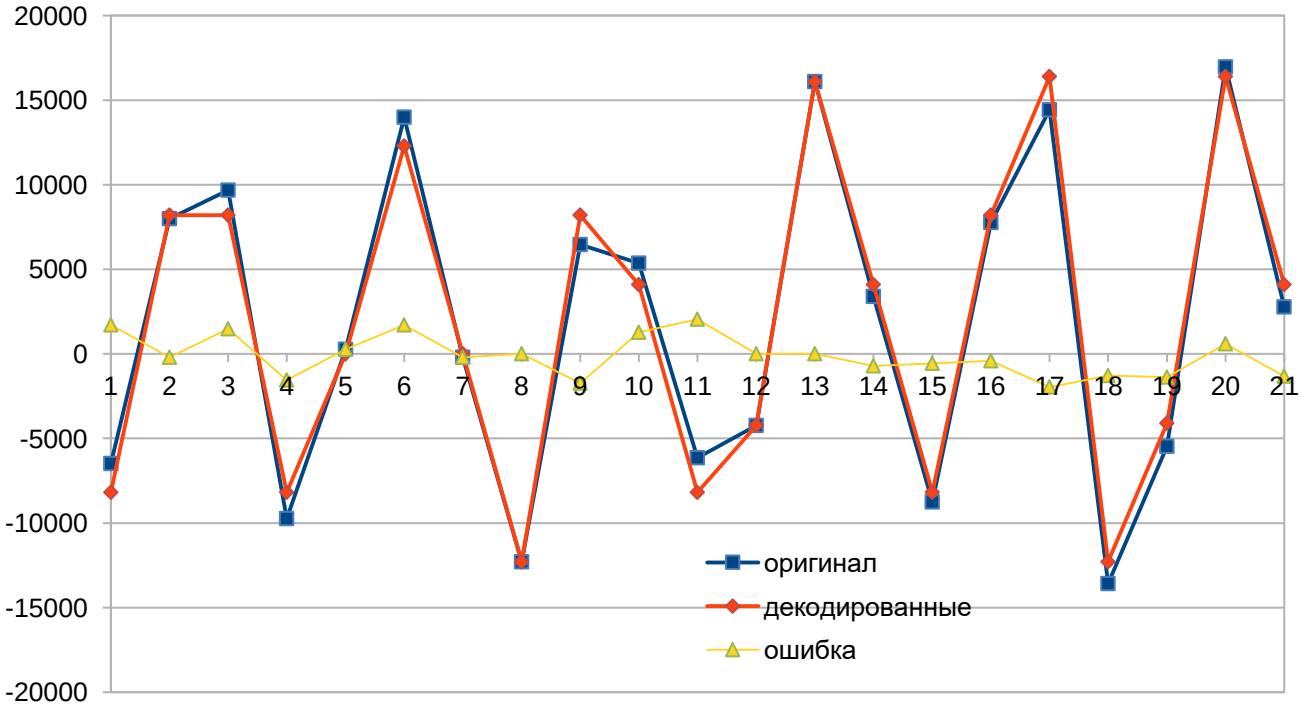


Рисунок 10 — Окрестности выборки с наибольшей абсолютной ошибкой

Среднеквадратичное отклонение — 337.2

Передескретизируем звуковой эффект на 8000 Гц

Качество звука заметно снизилось, но кодек не внес ощутимых искажений.

Наибольшая абсолютная ошибка — 2048 (33.3%)

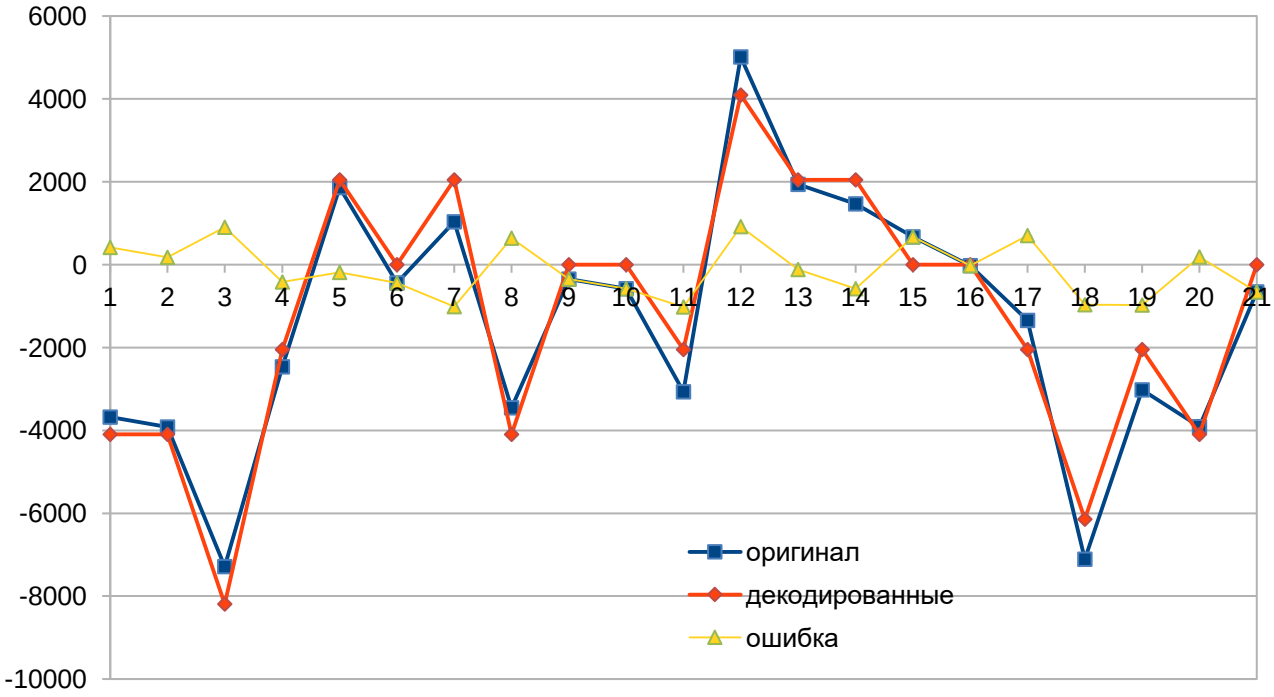


Рисунок 11 — Окрестности выборки с наибольшей абсолютной ошибкой

Среднеквадратичное отклонение — 183.6

Частота дискретизации, Гц	Наибольшая ошибка	СКО
44100	2048 (22,45%)	357,14
22050	2048 (33.3%)	337.2
8000	2048 (33.3%)	183.6

4.3 Исследование влияния ошибки на закодированный сигнал

Исходные данные: блок закодированных данных.

Вносились изменения вручную, в 1 бит подблока закодированных данных (19 байт, 152 бита, 32 выборки)

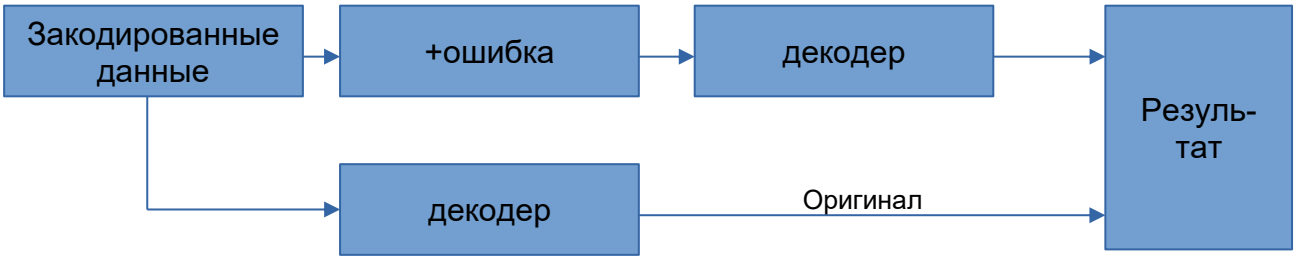


Рисунок 12 – Схема эксперимента

В следующие участки:

Заголовок:

а) незакодированные выборки (sample_0, sample_1) — старшие биты

а0) Первую (sample_0)

а1) Вторую (sample_1)

б) модель предсказания (coef_index)

в) экспоненту (exp_shift) - старший бит

Данные:

г) Первая закодированная выборка - старший бит

В одном подблоке каждая следующая выборка зависит от предыдушей, поэтому ошибка в одной выборке распространяется на последующие

Результат:

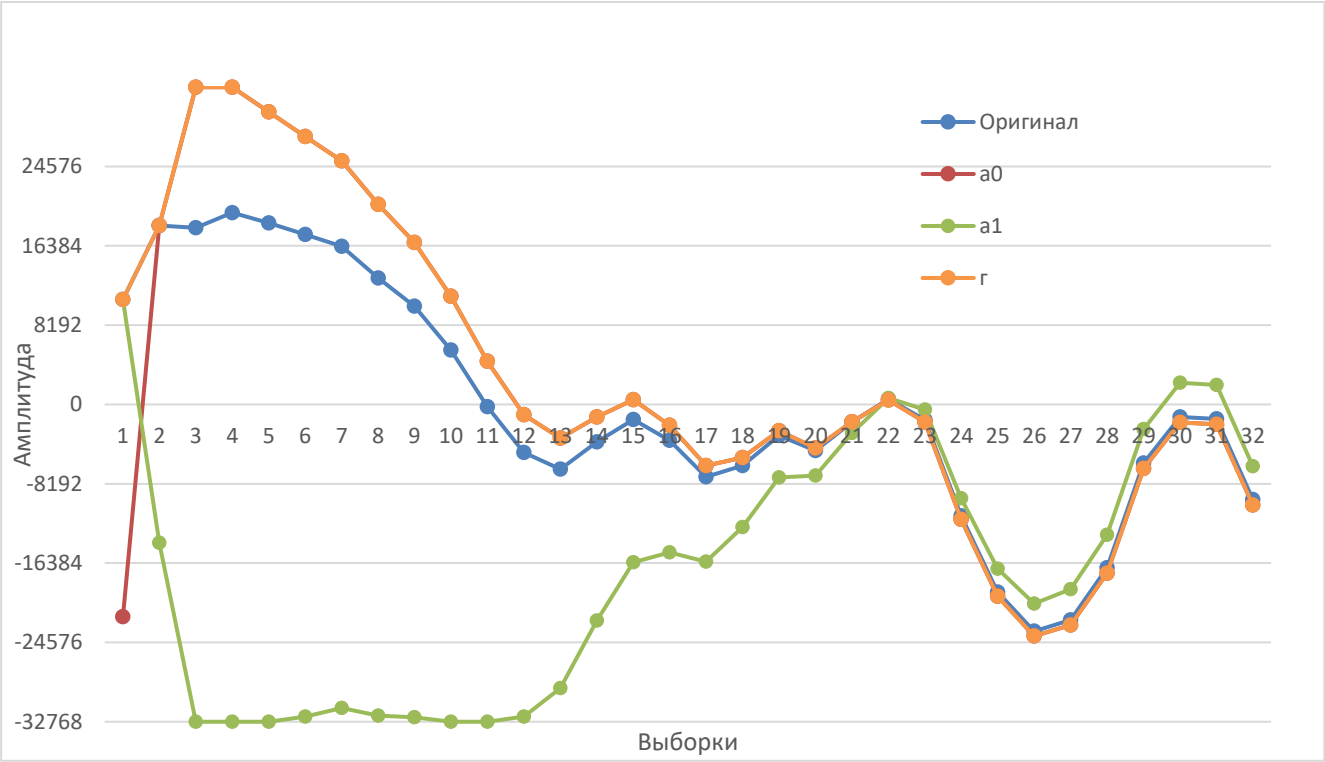


Рисунок 13 – Результат экспериментов а и г

В случаях а и г можно наблюдать уменьшение ошибки. В обоих случаях ошибка вносилась в выборки, поэтому результаты оказались похожи.

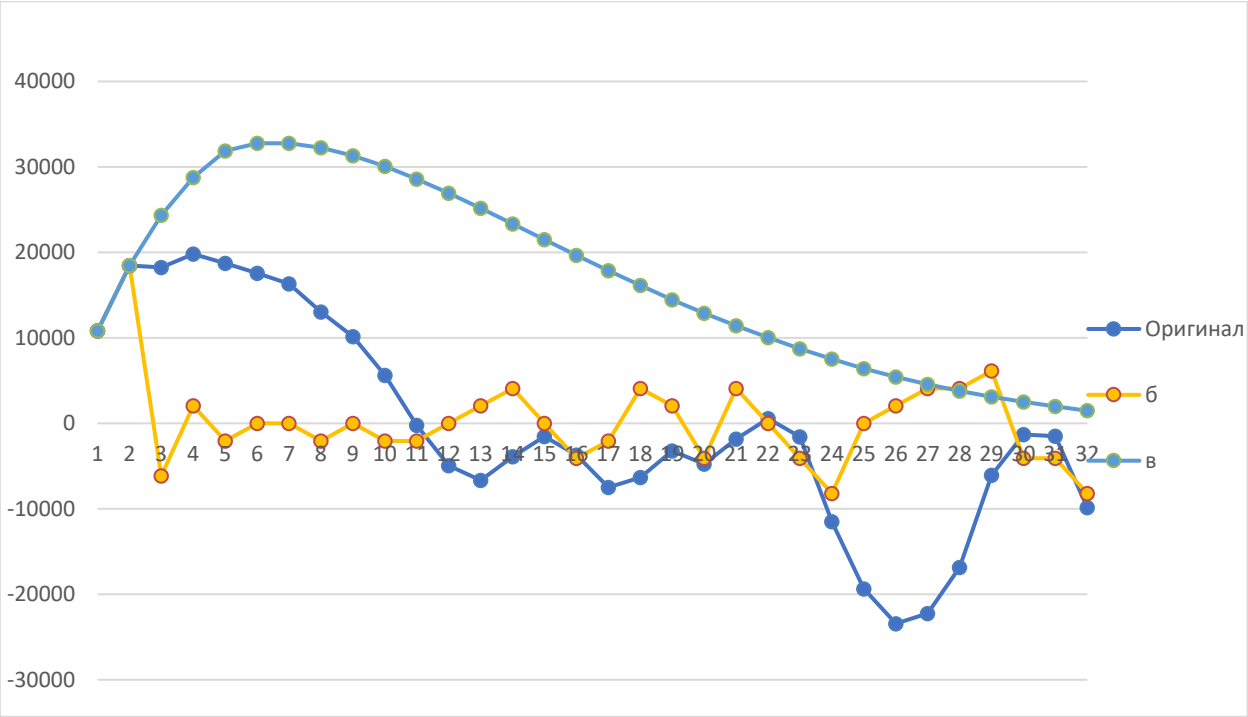


Рисунок 14 – Результат экспериментов б и в

б) Отражение волны с искажением

в) При декодировании использовалось практически только предсказание

Из результатов можно сделать вывод о том, что в случае повреждения выборок возможно самовосстановление, наиболее уязвимы к повреждению 6 бит из 152-х.

4.4 Преимущества и недостатки кодека

К преимуществам кодека можно отнести

- Быстроту кодирования/декодирования — на декодирование одной выборки нужно произвести 2 операции умножения, 3-4 операции сложения, 1-2 операции сдвига.

- Возможность оптимизации с помощью параллелизма, в то числе с помощью векторных SIMD инструкций.

- Постоянная степень сжатия, вне зависимости от поступивших данных, что является преимуществом в системах связи, однако может являться недостатком при ином применении, потому как не обеспечивает постоянного качества и соответственно не может использоваться для сжатия без потерь.

К недостаткам кодека можно отнести:

- Для кодирования модели предсказания достаточно 2-х бит, хотя зарезервировано 4 бита

- Перемешивание данных в блоке укрупняет блок, что может являться недостатком если нужно передать количество выборок менее 128. Так же это ограничивает декодирование блока по мере его получения.

ЗАКЛЮЧЕНИЕ

В проведенной работы был получен кодер/декодер ADPCM EA XAS, подходящий для вставок звуков в файлы игр, использующих данный кодек.

Можно сделать вывод, что кодек применим в системах спутниковой связи, если имеется необходимость и возможность ускорения кодирования/декодирования с помощью SIMD инструкций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. СТО 4.2-07-2010 "Система менеджмента качества. Общие требования к построению, изложению и оформлению документов учебной деятельности"

2. Hex-Rays – Interactive Disassembler (IDA) (англ.) [Электронный ресурс]: hex-rays.com

3. FFmpeg - (англ.) [Электронный ресурс]: ffmpeg.org

4. Использование дифференциальной импульсно-кодовой модуляции для кодирования звука / Д. В. Дорошев // Исследования и разработки в области машиностроения, энергетики и управления : материалы VI Междунар. межвуз. науч.-техн. конф. студентов, магистрантов иаспирантов, Гомель, 4–5 мая 2006 г. / М-во образования Респ. Беларусь, Гомел. гос. техн. ун-т им. П. О. Сухого. – Гомель : ГГТУ им. П. О. Сухого, 2006. – С. 393 - 396.

5. G.723 – ITU-T: 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)

6. Спилкер, Джеймс Дж. Цифровая спутниковая связь [Текст] / Пер. с англ. под ред. В.В. Маркова. - Москва : Связь, 1979. - 592 с.

ПРИЛОЖЕНИЕ А

Программный код: EA ADPCM codec.h

```
#pragma once

#include <string.h>
#include <iostream>

typedef unsigned char byte;

const int fixed_point_offset = 8;
const int fixp_exponent = 1 << fixed_point_offset;
/*
  S0 - prev-prev sample
  S1 - prev sample
  S2 - curr sample prediction
  dS = S1 - S0
*/

typedef int16_t table_type;
const table_type ea_adpcm_table_v2[][2] = {
  {(table_type)(0.000000*fixp_exponent), (table_type)(
0.000000*fixp_exponent)}, // S2  = 0
    silent,      also will used for high freq sound
  {(table_type)(0.937500*fixp_exponent), (table_type)(
0.000000*fixp_exponent)}, // S2 ~= 0.94*S1
    slight fading, also will used for high freq noise
  {(table_type)(1.796875*fixp_exponent), (table_type)(-
0.812500*fixp_exponent)}, // S2 ~= S1 + dS*0.8         = S0 + dS*1.8
  follow trend,  commonly used for low freq sound with high sample rate
  {(table_type)(1.531250*fixp_exponent), (table_type)(-
0.859375*fixp_exponent)}, // S2 ~= 0.67*S1 + 0.86*dS   = 0.67*S0 + dS*1.53
};


#ifdef __GNUC__
#include <x86intrin.h>
#define _byteswap_ushort __builtin_bswap16
#else

#ifndef _MSC_VER
#define _byteswap_ushort(VAL) (uint16_t)( VAL >> 8 | VAL << 8)
#endif // !_MSC_VER


#endif

const int subchunks_in_XAS_chunk = 4;
```

```
const int samples_in_XAS_subchunk = 30;
const int samples_in_XAS_header = 2;
const int samples_in_XAS_per_subchunk = samples_in_XAS_subchunk +
samples_in_XAS_header; // but not IN subchunk

const int samples_in_EA_XA_R_chunk = 28;
const int sizeof_EA_XA_R1_chunk = 1 + 2*sizeof(int16_t) +
samples_in_EA_XA_R_chunk / 2;
const int sizeof_uncompr_EA_XA_R23_block = 1 + (samples_in_EA_XA_R_chunk +
2) * sizeof(int16_t);
const int sizeof_compr_EA_XA_R23_block = 1 + samples_in_EA_XA_R_chunk / 2;

#pragma pack(push, 1)

// for x86, x64 MSVC!

// size 4 bytes, 2 samples
// Little Endian
struct XAS_SubChunkHeader {
  unsigned coef_index : 2;  // index for table for coeficien
  unsigned unused : 2;        // must be 0
  signed sample_0 : 12;
  unsigned exp_shift : 4;     // shift right, bits
  signed sample_1 : 12;
};

// MSVC thinking it's 4 bytes xD
struct SamplesByte {
  signed sample1 : 4;
  signed sample0 : 4;
};

struct SamplesDWORD {
  int16_t samples[2];
};

// size 76 bytes, 128 samples
struct XAS_Chunk {
  XAS_SubChunkHeader headers[subchunks_in_XAS_chunk]; // total size 16
bytes, 8 samples
  byte XAS_data[15][subchunks_in_XAS_chunk];     // data for each 2 samples
(1 bytes) interleaved, total size 60 bytes, 120 samples
};
#pragma pack(pop, 1)

#ifdef _MSC_VER

#include <intrin.h>
```

```
#define _memcpy(DST, SRC, _SIZE) __movsb((byte*)DST, (byte*)SRC, _SIZE)
#define _memset(DST, VAL, _SIZE) __stosb((byte*)DST, (byte)VAL, _SIZE)

#else // _MSC_VER

#define _memcpy memcpy
#define _memset memset

#endif


inline int16_t Get_s16be(const void* ptr) {
  return (short)_byteswap_ushort(*(unsigned short*)ptr);
}
inline int16_t bytestream2_get_le16s(byte** ptr) {
  short val = **(short**)ptr;
  *ptr += 2;
  return val;
}
inline char bytestream2_get_bytes(byte** ptr) {
  char val = **(char**)ptr;
  *ptr += 1;
  return val;
}
inline char low_sNibble(char _byte) {
  return (char)((byte)_byte << 4) >> 4;
}
inline int16_t Clip_int16(int val) {
#ifdef __SSE2__
  return _mm_cvtsi128_si32(_mm_packs_epi32(_mm_cvtsi32_si128(val),
_mm_undefined_si128()));
#else
  return (val >= 0x7FFF) ? 0x7FFF : (val <= -0x8000) ? -0x8000 : val;
#endif // __SSE2__
}
inline char Clip_int4(char val) {
  if (val >= 7) return 7;
  if (val <= -8) return -8;
  return val;
}
inline int Clip_fix_p16(int val) {
  if (val >= (0x7FFF << fixed_point_offset)) return 0x7FFF <<
fixed_point_offset;
  if (val <= (-0x8000 << fixed_point_offset)) return -0x8000 <<
fixed_point_offset;
  return val;
}
```

ПРИЛОЖЕНИЕ Б

Программный код: EA_ADPCM_DLL.h

```
#pragma once
#include <stdint.h>

// #define _DEBUG

#ifdef _MSC_VER

#ifdef EAADPCMCODEC_EXPORTS
#define EAADPCMCODEC_API extern "C" __declspec(dllexport)
#else
#define EAADPCMCODEC_API extern "C" __declspec(dllimport)
#endif
#define CODEC_ABI //__vectorcall

#else // _MSC_VER

#define EAADPCMCODEC_API
#define CODEC_ABI

#endif

EAADPCMCODEC_API
uint32_t CODEC_ABI GetXASEncodedSize(uint32_t n_samples_per_channel,
uint32_t n_channels);

EAADPCMCODEC_API
void CODEC_ABI decode_XAS(const void* in_XAS, int16_t* out_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels);

EAADPCMCODEC_API
void CODEC_ABI encode_XAS(void* out_XAS, const int16_t* in_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels);

EAADPCMCODEC_API
void CODEC_ABI decode_EA_XA_R2(const void* data, int16_t *out_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels);

EAADPCMCODEC_API
size_t CODEC_ABI encode_EA_XA_R2(void* data, const int16_t PCM[], uint32_t
n_samples_per_channel, uint32_t n_channels, int16_t max_error = 10);


// #define BENCH

#ifdef BENCH
```

```
EAADPCMCODEC_API
#ifdef _MSC_VER
void _cdecl Bench(uint32_t reps);
#endif
void CODEC_ABI Bench(uint32_t reps);

#endif // !BENCH
```

ПРИЛОЖЕНИЕ В

Программный код: EA-ADPCM-codec.cpp

```
#include "limits.h"

#include "vector_SIMD.h"

#include "EA ADPCM codec.h"
#include "EA_ADPCM_DLL.h"

#include <cassert>

#ifdef _DEBUG
#include <iostream>
#endif // _DEBUG

/*
  bias_compens
  EA-XA R2: not present in ffmpeg and SX, but present in NFS_abk_decode
  EA-XAS: presents in all known decoders (ffmpeg, NFS_abk_decoder(Carbon+))
*/
const int def_rounding = (fixp_exponent >> 1);

inline int16_t decode_XA_sample(const int16_t prev_samples[2], const
table_type coef[2], char int4, byte shift) {
  int correction = (int)int4 << shift;
  int prediction = prev_samples[1] * coef[0] + prev_samples[0] * coef[1];
  return Clip_int16((prediction + correction + def_rounding) >>
fixed_point_offset);
}

struct EncodedSample {
  int16_t decoded;
  char encoded;
};

inline EncodedSample encode_XA_sample(const int16_t prev_samples[2], const
table_type coef[2], int sample, byte shift) {
  int prediction = prev_samples[1] * coef[0] + prev_samples[0] * coef[1];

  int correction = (sample << fixed_point_offset) - prediction;

#ifdef _DEBUG__
    int shifted = correction >> shift;
        const int tr = 8;
        if (shifted > tr || shifted < -(tr + 1)) {
          shift = out_chunk->headers[j].exp_shift - 1;
```

```
#ifdef _DEBUG
           printf("patch used for sample %d\n", pInSamples - in_PCM);
#endif
           // goto patch;
         }
#endif

    int res;
    int rounding = 1 << (shift - 1);
    res = Clip_int4((correction + rounding) >> shift);

  int predecoded = ((res << shift) + prediction + def_rounding) >>
fixed_point_offset;
  int decoded = Clip_int16(predecoded);

  // ---- for better precision on clipping or near-clipping, this can be
removed
  int term = 1 << (shift - fixed_point_offset); // it's like +-1 to res
until >> fixed_point_offset
  int decoded2;
    decoded2 = Clip_int16(predecoded + term);
  if (res != 7 && abs(decoded - sample) > abs(decoded2 - sample)) {
    res += 1;
    decoded = decoded2;
  }
  else {
        decoded2 = Clip_int16(predecoded - term);
        if (res != -8 && abs(decoded - sample) > abs(decoded2 - sample)) {
            res -= 1;
            decoded = decoded2;
        }
    }
    // ----

  return { (int16_t)decoded, (char)res};
}

#define _GetNumXASChunks(N_SAMPLES) ((N_SAMPLES + 127) / 128)
// mb export?
uint32_t GetNumXASTotalChunks(uint32_t n_samples_per_channel, uint32_t
n_channels) {
  return n_channels * _GetNumXASChunks(n_samples_per_channel);
}

EAADPCMCODEC_API
uint32_t GetXASEncodedSize(uint32_t n_samples_per_channel, uint32_t
n_channels) {
  return GetNumXASTotalChunks(n_samples_per_channel,
n_channels)*sizeof(XAS_Chunk);
```

```
}

void decode_XAS_Chunk(const XAS_Chunk* in_chunk, int16_t* out_PCM) {
  for (int j = 0; j < subchunks_in_XAS_chunk; j++) {
    int16_t *pSamples = out_PCM + j * 32;

    pSamples[0] = (in_chunk->headers[j].sample_0 << 4);
    int coef_index = in_chunk->headers[j].coef_index;

    pSamples[1] = (in_chunk->headers[j].sample_1 << 4);
    byte shift = 12 + fixed_point_offset - in_chunk->headers[j].exp_shift;

    const table_type* coef = ea_adpcm_table_v2[coef_index];

    for (int i = 0; i < 15; i++, pSamples += 2) {
      SamplesByte data = *(SamplesByte*)&(in_chunk->XAS_data[i][j]);

      pSamples[2] = decode_XA_sample(pSamples, coef, data.sample0, shift);
      pSamples[3] = decode_XA_sample(pSamples + 1, coef, data.sample1,
shift);

#ifdef _DEBUG
      EncodedSample enc0 = encode_XA_sample(pSamples, coef, pSamples[2],
shift);
      EncodedSample enc1 = encode_XA_sample(pSamples + 1, coef,
pSamples[3], shift);
      if (enc0.encoded != data.sample0 || enc1.encoded != data.sample1) {
        printf(__FUNCTION__ " subchunk %d, byte %d: ", j, i);
        if (enc0.decoded != pSamples[2] || enc0.decoded != pSamples[3]) {
          printf("reencode error \n");
        }
        else {
          printf("loseless reencoding unequality \n");
        }
      }
#endif // _DEBUG
    }
  }
}

#ifdef __GNUC__
#define ALIGN(ALGN) __attribute__((aligned (ALGN)))
#else
#define ALIGN(ALGN)
#endif

void decode_XAS_Chunk_SIMD(const XAS_Chunk* in_chunk, int16_t* out_PCM) {
    vec128 head = LoadUnaligned(in_chunk->headers);
  static const table_type ea_adpcm_table_v3[][2] ALIGN(16) = {
```

```
    {(table_type)(0.000000*fixp_exponent),
(table_type)(0.000000*fixp_exponent)},
    {(table_type)(0.000000*fixp_exponent),
(table_type)(0.937500*fixp_exponent)},
    {(table_type)(-0.812500*fixp_exponent),
(table_type)(1.796875*fixp_exponent)},
    {(table_type)(-0.859375*fixp_exponent),
(table_type)(1.531250*fixp_exponent)},
  };
  static const int32_t const_shift[4] ALIGN(16) = {16 - fixed_point_offset,
16 - fixed_point_offset , 16 - fixed_point_offset , 16 - fixed_point_offset
};
  static const uint8_t shuffle[16] ALIGN(16) = {12, 8, 4, 0,   13, 9, 5, 1,
14, 10, 6, 2,    15, 11, 7, 3};

  uint32x4_t rounding = { GetOnes128() };

  uint32x4_t coef_mask = rounding >> 30;
  int32x4_t nibble_mask = rounding << 28;

  rounding = (rounding >> 31 << (fixed_point_offset -
1)).SIMD_reinterpret_cast<uint32x4_t>();

  int16x8_t samples = head.SIMD_reinterpret_cast<int16x8_t>();
  samples = samples >> 4 << 4;

  int32x4_t shift = { head };
  shift = *(int32x4_t*)const_shift + ((shift <<
12).SIMD_reinterpret_cast<uint32x4_t>() >> 28);
  int32x4_t coef_index = { head & coef_mask };
  int16x8_t coefs = LoadByIndex(coef_index, (int*)
ea_adpcm_table_v3).SIMD_reinterpret_cast<int16x8_t>();

    SaveWithStep(samples.SIMD_reinterpret_cast<int32x4_t>(),
(int32_t*)out_PCM, 16);

  vec128 _shuffle = *(vec128*)shuffle;

  for (int i = 0; i < 4; i++) {

        int32x4_t data = LoadUnaligned(&in_chunk-
>XAS_data[0][i*16]).SIMD_reinterpret_cast<int32x4_t>();

    data = PermuteByIndex(data,
_shuffle).SIMD_reinterpret_cast<int32x4_t>();

    int itrs = 4 - ((i + 1) >> 2); // i != 3 ? 4 : 3;

    for (int j = 0; j < itrs; j++) {
```

```
      for (int k = 0; k < 2; k++) {
        int32x4_t prediction = mul16_add32(samples, coefs);
        int32x4_t correction = (data &
nibble_mask).SIMD_reinterpret_cast<int32x4_t>() >> shift;

        int32x4_t predecode = (prediction + correction + rounding) >>
fixed_point_offset;

        int16x8_t decoded = Clip_int16(predecode);

        samples = { (samples.SIMD_reinterpret_cast<uint32x4_t>() >> 16) |
(((int32x4_t)(decoded.SIMD_reinterpret_cast<uint16x8_t>())) << 16) };

        data = data << 4;
      }
      SaveWithStep(samples.SIMD_reinterpret_cast<int32x4_t>(),
(int*)(out_PCM + i*8 + j*2 + 2), 16);
    }
  }
#ifdef _DEBUG
  int16_t PCM2[128];
  decode_XAS_Chunk(in_chunk, PCM2);
  if (memcmp(PCM2, out_PCM, 128 * 2) == 0) {
    printf("ok \n");
  }
  else {
    printf("not ok \n");
  }
#endif // _DEBUG
}

#ifdef BENCH
void PrintRes(const char* mes, uint64_t time, uint64_t reps) {
  printf("%s: total = %llu, per chunk = %f \n", mes, time, (double)time /
reps);
}
void Bench(uint32_t reps) {
  XAS_Chunk in_chunk;
  int16_t PCM[128];
  uint64_t start = __rdtsc();
  for (uint64_t i = 0; i < reps; i++) {
    decode_XAS_Chunk(&in_chunk, PCM);
  }
  uint64_t SISD_time = __rdtsc() - start;
  start = __rdtsc();
  for (uint64_t i = 0; i < reps; i++) {
    decode_XAS_Chunk_SIMD(&in_chunk, PCM);
  }
  uint64_t SIMD_time = __rdtsc() - start;
```

```
  PrintRes("SISD", SISD_time, reps);
  PrintRes("SIMD", SIMD_time, reps);
}
#endif // BENCH

#define decode_XAS_Chunk decode_XAS_Chunk_SIMD

void decode_XAS(const void* in_data, int16_t* out_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels) {
  if (n_samples_per_channel == 0)
    return;
  const XAS_Chunk* _in_data = (XAS_Chunk*)in_data;
  int16_t PCM[128];
  uint32_t n_chunks_per_channel = _GetNumXASChunks(n_samples_per_channel);
  for (int chunk_ind = 0; chunk_ind < n_chunks_per_channel - 1;
chunk_ind++) {
    for (int channel_ind = 0; channel_ind < n_channels; channel_ind++) {
      decode_XAS_Chunk(_in_data++, PCM);
      for (int sample_ind = 0; sample_ind < 128; sample_ind++) {
        out_PCM[channel_ind + sample_ind * n_channels] = PCM[sample_ind];
      }
    }
    out_PCM += 128* n_channels;
  }
  uint32_t samples_remain_per_channel = n_samples_per_channel -
(n_chunks_per_channel-1)*128;
  for (int channel_ind = 0; channel_ind < n_channels; channel_ind++) {
    decode_XAS_Chunk(_in_data++, PCM);
    for (int sample_ind = 0; sample_ind < samples_remain_per_channel;
sample_ind++) {
      out_PCM[channel_ind + sample_ind * n_channels] = PCM[sample_ind];
    }
  }
}

// ~same method as in SX but with fixed point
int simple_CalcCoefShift(const int16_t* pSamples, const int16_t
in_prevSamples[2], int num_samples, int *out_coef_index, byte* out_shift) {
  // SX using clip here

  const int num_coefs = 4;

  int min_max_error = INT_MAX;
  int s_min_max_error = INT_MAX; // don't need I think
  int best_coef_ind = 0;
  for (int coef_ind = 0; coef_ind < num_coefs; coef_ind++) {
    int16_t prevSamples[2] = { in_prevSamples[0], in_prevSamples[1] };
    // fixed point 24.8
```

```
    // for coef_ind = 0 max_error = max abs sample
    int max_error = 0;
    int s_max_error = 0;
    for (int i = 0; i < num_samples; i++) {
      int prediction = ea_adpcm_table_v2[coef_ind][0] * prevSamples[1] +
ea_adpcm_table_v2[coef_ind][1] * prevSamples[0];
      int sample = pSamples[i];
      sample <<= fixed_point_offset;
      int s_error = sample - prediction;
      int error = abs(s_error);
      if (error > max_error) {
        max_error = error;
        s_max_error = s_error;
      }
      prevSamples[0] = prevSamples[1];
      prevSamples[1] = pSamples[i];
    }
    if (max_error < min_max_error) {
      min_max_error = max_error;
      best_coef_ind = coef_ind;
      s_min_max_error = s_max_error;
    }
  }
  int max_min_error_i16 = Clip_int16(min_max_error >> fixed_point_offset);

  int mask = 0x4000;
  int exp_shift;
  for (exp_shift = 0; exp_shift < 12; exp_shift++) {
    if ((((mask >> 3) + max_min_error_i16) & mask) != 0) {
      break;
    }
    mask >>= 1;
  }
  *out_coef_index = best_coef_ind;
  *out_shift = exp_shift;
  return max_min_error_i16;
}


const int shift4_rounding = 0x8 - 1;

void encode_XAS_Chunk(XAS_Chunk* out_chunk, const int16_t in_PCM[128] /*,
size_t nSamples = 128*/) {
  //assert(nSamples <= 128);
  for (int j = 0; j < subchunks_in_XAS_chunk; j++) {

    const int16_t *pInSamples = in_PCM + j * 32;

        out_chunk->headers[j].unused = 0;
```

```
    out_chunk->headers[j].sample_0 = (pInSamples[0] + shift4_rounding) >>
4;
    out_chunk->headers[j].sample_1 = (pInSamples[1] + shift4_rounding) >>
4;

    int16_t decoded_PCM[32];
    decoded_PCM[0] = out_chunk->headers[j].sample_0 << 4;
    decoded_PCM[1] = out_chunk->headers[j].sample_1 << 4;

    int coef_index;
    byte shift;
    simple_CalcCoefShift(pInSamples + 2, decoded_PCM, 30, &coef_index,
&shift);
patch:
    out_chunk->headers[j].coef_index = coef_index;
    out_chunk->headers[j].exp_shift = shift;

    const table_type *coef = ea_adpcm_table_v2[coef_index];
    shift = 12 + fixed_point_offset - shift;

    int16_t *pDecodedSamples = decoded_PCM;

    for (int i = 0; i < 15; i++) {
      byte data = 0;

      for (int n = 0; n < 2; n++) {
        EncodedSample enc = encode_XA_sample(pDecodedSamples, coef,
pInSamples[2], shift);

        pDecodedSamples[2] = enc.decoded; // think as decoder will for
better precision
        data <<= 4;
        data |= enc.encoded & 0xF;
        pInSamples++, pDecodedSamples++;
      }
      out_chunk->XAS_data[i][j] = data;
    }
  }
}


void encode_XAS(void* out_data, const int16_t* in_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels) {
  if (n_samples_per_channel == 0)
    return;
  XAS_Chunk* _out_data = (XAS_Chunk*)out_data;
  uint32_t n_chunks_per_channel = _GetNumXASChunks(n_samples_per_channel);
  int16_t PCM[128];
#pragma nounroll
```

```
  for (int chunk_ind = 0; chunk_ind < n_chunks_per_channel - 1;
chunk_ind++) {
    for (int channel_ind = 0; channel_ind < n_channels; channel_ind++) {
      const int16_t* t = in_PCM + channel_ind;
      for (int sample_ind = 0; sample_ind < 128; sample_ind++, t +=
n_channels) {
        PCM[sample_ind] = *t;
      }
      encode_XAS_Chunk(_out_data++, PCM);

    }
    in_PCM += 128 * n_channels;
  }
  uint32_t samples_remain_per_channel = n_samples_per_channel -
(n_chunks_per_channel - 1) * 128;
  for (int channel_ind = 0; channel_ind < n_channels; channel_ind++) {
    for (int sample_ind = 0; sample_ind < samples_remain_per_channel;
sample_ind++) {
      PCM[sample_ind] = in_PCM[channel_ind + sample_ind * n_channels];
    }
    _memset(PCM + samples_remain_per_channel, 0, (128 -
samples_remain_per_channel)*sizeof(int16_t));
    encode_XAS_Chunk(_out_data++, PCM);
  }
}


// processing 28 samples, returns number of bytes red (61 or 15)
size_t decode_EA_XA_R2_Chunk(const byte* XA_Chunk, int16_t out_PCM[28],
int16_t prev_samples[3]) {
  const byte* p_curr_byte = XA_Chunk;
  int16_t *pSample = out_PCM;
  byte _byte = *(p_curr_byte++);
  int16_t* p_prev_samples = prev_samples;
  if (_byte == 0xEE) {
    prev_samples[1] = Get_s16be(p_curr_byte), p_curr_byte += 2;
    prev_samples[0] = Get_s16be(p_curr_byte), p_curr_byte += 2;
    for (int i = 0; i < samples_in_EA_XA_R_chunk; i++)
      *(pSample++) = Get_s16be(p_curr_byte), p_curr_byte += 2;
  }
  else {
    int coef_index = _byte >> 4;
    const table_type *coef = ea_adpcm_table_v2[coef_index];
    byte shift = 12 + fixed_point_offset - (_byte & 0xF);
    for (int j = 0; j < samples_in_EA_XA_R_chunk / 2; j++) {
      SamplesByte data = *(SamplesByte*)(p_curr_byte++);

      pSample[0] = decode_XA_sample(p_prev_samples, coef, data.sample0,
shift);
```

```
      prev_samples[2] = pSample[0]; // in case of p_prev_samples ==
prev_samples
      pSample[1] = decode_XA_sample(p_prev_samples + 1, coef,
data.sample1, shift);

#ifdef _DEBUG
      EncodedSample enc = encode_XA_sample(p_prev_samples + 1, coef,
pSample[1], shift);
      if (enc.encoded != data.sample1) {
        printf(
          "Reencoding issue:\n"
          "   source sample = %d (0x%X)\n"
          "reencoded sample = %d (0x%X)\n"
          "   source nibble = %d (0x%X)"
          "reencoded nibble = %d (0x%X)\n\n",
          (int)pSample[1], (int)pSample[1], (int)enc.decoded,
(int)enc.decoded,
          (int)data.sample1, (int)data.sample1, (int)enc.encoded,
(int)enc.encoded);
      }
#endif // _DEBUG


      p_prev_samples = pSample;
      pSample += 2;
    }
    prev_samples[1] = pSample[-1];
    prev_samples[0] = pSample[-2];
  }

  return p_curr_byte - XA_Chunk;
}

void decode_EA_XA_R2(const void* data, int16_t *out_PCM, uint32_t
n_samples_per_channel, uint32_t n_channels) {
    // TODO: multi channel
    byte *_data = (byte*)data;
  int16_t prev_samples[3] = { 0 };
    int num_chunks = (n_samples_per_channel + 27) / 28;
  for (int i = 0; i < num_chunks; i++) {
    size_t data_decoded_size = decode_EA_XA_R2_Chunk(_data, out_PCM,
prev_samples);
#ifdef _DEBUG
    if (data_decoded_size != sizeof_uncompr_EA_XA_R23_block
      && data_decoded_size != sizeof_compr_EA_XA_R23_block) {
      printf("Warning: decoded %d bytes\n", data_decoded_size);
      system("pause");
    }
#endif // _DEBUG
```

```
        _data += data_decoded_size;
    out_PCM += samples_in_EA_XA_R_chunk;
  }
}

void encode_EA_XA_R2_chunk_nocompr(byte
data[sizeof_uncompr_EA_XA_R23_block], const int16_t PCM[28], int16_t
prev[2], int nCannels) {
  *data = 0xEE;
    *(int16_t*)(data + 1) = _byteswap_ushort(PCM[26*nCannels]);
    *(int16_t*)(data + 3) = _byteswap_ushort(PCM[27*nCannels]);
    prev[0] = PCM[26*nCannels];
    prev[1] = PCM[27*nCannels];
  int16_t* pOutData = (int16_t*)(data + 5);
  for (int i = 0; i < 28*nCannels; i+=nCannels) {
    pOutData[i] = _byteswap_ushort(PCM[i]);
  }
}

void encode_EA_XA_block(byte data[], const int16_t PCM[], int16_t prev[2],
int samples, int PCM_step, const table_type* coefs, byte shift, int
data_step = 1){
    for (int i = 0; i < samples/2; i++){
        byte _data = 0;
        for (int j = 0; j < 2; j++){
            EncodedSample enc = encode_XA_sample(prev, coefs, PCM[(i*2 +
j)*PCM_step], shift);
            prev[0] = prev[1];
            prev[1] = enc.decoded;
            _data <<= 4;
            _data |= enc.encoded;
        }
        *data = _data;
        data += data_step;
    }
}

size_t encode_EA_XA_R2_chunk(byte data[sizeof_uncompr_EA_XA_R23_block],
const int16_t PCM[28], int16_t prev[2], int nCannels, int16_t max_error) {
    int coef_index;
    byte shift;
    int err = simple_CalcCoefShift(PCM, prev, 28, &coef_index, &shift);
    if (err > max_error){
        encode_EA_XA_R2_chunk_nocompr(data, PCM, prev, nCannels);
        return sizeof_uncompr_EA_XA_R23_block;
    }
    else {
        *data++ = coef_index << 4 | shift;
        shift = 12 + fixed_point_offset - shift;
```

```
        const table_type * coefs = ea_adpcm_table_v2[coef_index];
        encode_EA_XA_block(data, PCM, prev, 28, nCannels, coefs, shift);
        return sizeof_compr_EA_XA_R23_block;
    }
}

void encode_EA_XA_R1_chunk(byte data[sizeof_EA_XA_R1_chunk], const int16_t
PCM[28], const int16_t prev[2],  int nCannels) {
    *(int16_t*)data = _byteswap_ushort(prev[0]);
    *(int16_t*)(data + 2) = _byteswap_ushort(prev[1]); // ?
    int coef_index;
    byte shift;
    simple_CalcCoefShift(PCM, prev, 28, &coef_index, &shift);
    data[4] = coef_index << 4 | shift;
    int16_t _prev[2]; memcpy(_prev, prev, 4);
    encode_EA_XA_block(data + 5, PCM, _prev, 28, nCannels,
ea_adpcm_table_v2[coef_index], 12 + fixed_point_offset - shift);
}

size_t encode_EA_XA_R2_channel(void* data, const int16_t PCM[], uint32_t
n_samples_per_channel, uint32_t n_channels, int16_t max_error) {
    int chunks_per_channel = (n_samples_per_channel + 27) / 28;
    int16_t prev[2];
    byte* curr_data = (byte*)data;
    encode_EA_XA_R2_chunk_nocompr(curr_data, PCM, prev, (int)n_channels);
    curr_data += sizeof_uncompr_EA_XA_R23_block;
    for (int chunk_ind = 1; chunk_ind < chunks_per_channel; chunk_ind++){
        curr_data += encode_EA_XA_R2_chunk(curr_data, PCM +
28*chunk_ind*n_channels, prev, n_channels, max_error);
    }
    return curr_data - (byte*)data;
}

size_t encode_EA_XA_R2(void* data, const int16_t PCM[], uint32_t
n_samples_per_channel, uint32_t n_channels, int16_t max_error) {
    byte* curr_data = (byte*)data;
    for (int chan_ind = 0; chan_ind < n_channels; chan_ind++){
        curr_data += encode_EA_XA_R2_channel(curr_data, PCM + chan_ind,
n_samples_per_channel, n_channels, max_error);
    }
    return curr_data - (byte*)data;
}
```

Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой
_____О.В.Непомнящий
«20» 06 2022 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 «Информатика и вычислительная техника»

Разработка и исследование голосового кодека для возможности применения в спутниковой связи

Руководитель _____ _____ проф., д-р техн. наук С. А. Бронов
подпись *дата* *должность, ученая степень*

Выпускник _____ _____ П. В. Винокуров
подпись *дата*

Нормоконтролер _____ _____ проф., д-р техн. наук С. А. Бронов
подпись *дата* *должность, ученая степень*

Красноярск 2022