УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

НОВИ САД

Департман за рачунарство и аутоматику

Одсек за рачунарску технику и рачунарске комуникације

ПРОЈЕКТНИ ЗАДАТАК

Кандидат: **Павле Вуковић**

Број индекса: **RA135/2019**

Предмет: **Основи алгоритама и структура ДСП-а 1**

Тема рада: **Пројектни задатак 2**

Нови Сад, децембар, 2021.

Садржај

Увод

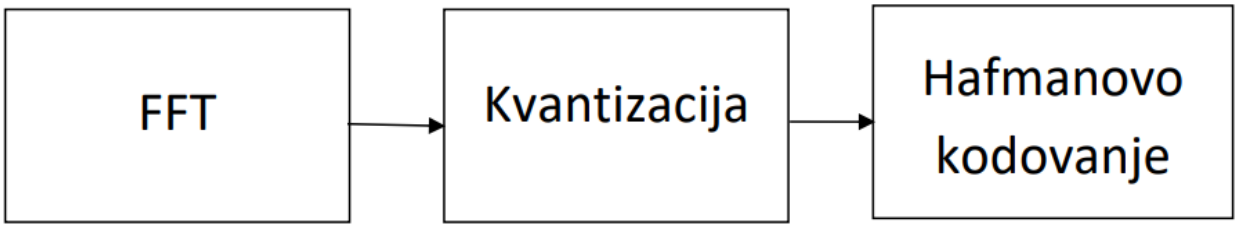
Пројектни задатак 2 је имплементација компресије аудио сигнала у програмском језику **C**. Компресија је смањење величине аудио сигнала на неко одређено време, како би се сигнал као такав могао лакше преносити и обрађивати. Након компресије намеће се фаза декомпресије, у којој се сигнал враћа у почетно стање. То је фаза у потпуности инверзна претходној. У зависности од тога који алгоритам компресије и декомпресије користимо, могуће је да излазни сигнал буде мање или више веродостојан у односу на улазни. Нама је у интересу наравно да након компресије добијемо компресовани фајл најмање могуће величине, а да након декомпресије добијемо фајл најбољег могућег квалитета. Различити поступци компресије/декомпресије имају различите односе величина компресованог фајла-квалитет излазног фајла.

1.Компресија((Ен)кодовање)

У почетку смо користили једноставан алгоритам за кодовање, састављен из фаза:

Прелазак у фреквентни домен -> **Квантизација** -> **Хафманово кодовање**,

Као што се може видети на слици 1.



Слика 1. Фазе кодовања

Наиме, након преузимања садржаја левог и десног канала улазног фајла, који је у мом случају био “15.wav”, долази до позива функције **void encode(Int16* inL, Int16* inR, Int16* out, Int16 N, Uint16 BL, Uint16 BR)**, где је идеја да се у датој функцији имплементира горе наведени алгоритам кодовања. Параметри функције су:

inL - InputBufferL - леви улазни канал;

inR - InputBufferR - десни улазни канал;

out - CompressedBuffer - јединствени излазни канал;

N - AUDIO_IO_SIZE - МАКРО, дужине 128, представља дужину улазних канала;

BL - број бита за квантизацију левог канала;

BR - број бита за квантизацију десног канала.

1.1 Прелазак у фреквентни домен(FFT)

Имплементиран је коришћењем брзе Фуријеове трансформације. Користи се метода преклопи и сабери, коришћена и на вежбама, како би каснија реконструкција била што боља. Имплементација овог дела енкодера се налази у оквиру горепоменуте функције encode(), а њен изглед је следећи:

```
63    //*****************************************************************************************************
64    //FFT
65
66    //Za LEVI kanal
67
68    for(i = 0; i < N; i++){
69        fft_bufferL[i] = in_delayL[i];
70        fft_bufferL[i + N] = inL[i];
71        in_delayL[i] = inL[i];
72    }
73
74    for(i = 0; i < 2*N; i++){
75        fft_bufferL[i] = _smpy(fft_bufferL[i], window[i]);
76    }
77
78    rfft(fft_bufferL, FFT_SIZE, SCALE);
79
80
81
82    //Za DESNI kanal
83
84    for(i = 0; i < N; i++){
85        fft_bufferR[i] = in_delayR[i];
86        fft_bufferR[i + N] = inR[i];
87        in_delayR[i] = inR[i];
88    }
89
90    for(i = 0; i < 2*N; i++){
91        fft_bufferR[i] = _smpy(fft_bufferR[i], window[i]);
92    }
93
94    rfft(fft_bufferR, FFT_SIZE, SCALE);
95    //*****************************************************************************************************
```

Слика 2. Имплементација FFT

1.2 Квантизација

Заснована је на функцији **Int16 quantB(Int16 input, Uint16 B);** Она над сваким одбирком врши квантизацију по амплитуди за задати број бита. Имплементација овог дела енкодера се налази у оквиру горепоменуте функције encode(), а њен изглед је следећи:

```
Int16 quantB(Int16 input, Uint16 B)
{
    Int16 output;
    Int16 round_factor = (1<<(16-B -1));

    if(INT16_MAX - round_factor < input)
    {
        output = (INT16_MAX >> (16-B));
    }
    else
    {
        output = ((input+round_factor) >> (16-B));
    }

    return output;
}
```

Слика 3. Функција quantB()

```
        //*****************************************************************************************************
        //KVANTIZACIJA

        //Za LEVI kanal

        for(i = 0; i < 2*N; i++){
            fft_bufferL[i] = quantB(fft_bufferL[i], BL);
        }



        //Za DESNI kanal

        for(i = 0; i < 2*N; i++){
            fft_bufferR[i] = quantB(fft_bufferR[i], BR);
        }
    }
    //*****************************************************************************************************
```

Слика 4. Део кода који врши квантизацију

1.3 Учешљавање

Након квантизације, ми у привременим каналима **fft_bufferL** и **fft_bufferR** поседујемо обрађене леви и десни улазни канал. Циљ је да такве новодобијене канале насумично увежемо у један излазни канал - **CompressedBuffer**. То радимо тако што његов садржај попуњавамо насумично вредностима левог и десног бафера. Имплементација овог дела енкодера се налази у оквиру горепоменуте функције encode(), а њен изглед је следећи:

```
    //*****************************************************************************************************
    //UCESLJAVANJE

    for(i = 0; i < 2*N; i++){
        out[2*i] = fft_bufferL[i];
        out[2*i + 1] = fft_bufferR[i];
    }
    //*****************************************************************************************************
```

Слика 5. Део кода који врши учешљавање

1.4 Хафманово кодовање

За наше потребе је реализована апликација са називом **HuffmanEnc,** која врши Хафманово кодовање уместо нас. Она нас доводи до нашег коначног циља, а то је коначни компресовани фајл. Команда која позива ову апликацију је следећа:

HuffmanEnc <input_file> <compressed_file> <dictionary> ,

Јако је важно да се првенствено позиционирамо у директоријум где се ова функција налази, а након тога да одредимо одговарајуће улазне и излазне параметре. Ова функција ће на основу фајла ког добије из енкодера, који је у формату .dsp1, генерисати компресовани фајл и речник, који представља начин кодовања, који ће се касније користити као кључ за декодовање.

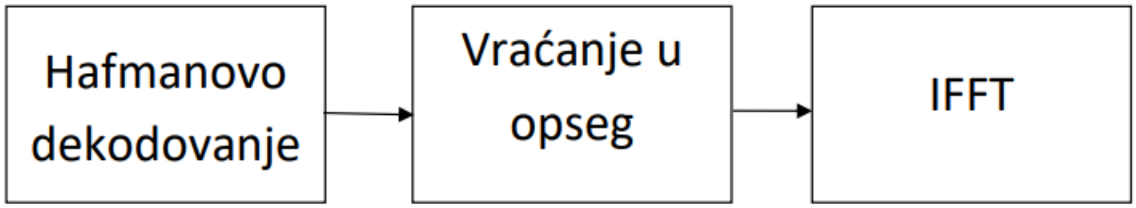
****

2.Декомпресија(Декодовање)

У почетку смо користили једноставан алгоритам за декодовање, састављен из фаза:

Хафманово декодовање -> **Реконструкција** -> **Прелазак у временски домен**

Као што се може видети на слици 6.



Слика 6. Фазе декодовања

2.1 Хафманово декодовање

На самом почетку декодовања се врши Хафманово декодовање. То је процес инверзан Хафмановом кодовању, што значи да он компресовани фајл пребацује у исти онај фајл који је био излаз из енкодера, где се тај .dsp1 фајл користи за улаз у декодер који је имплементирам у оквиру алата CCS. Хафманово декодовање се покреће позивом следеће команде:

```
HuffmanDec <compressed_file> <output_file> <dictionary>
```

Где је улазни фајл - компресовани фајл из излаза Хафмановог енкодера, као и речник из излаза Хафмановог енкодера, а излазни фајл је, као што је горе речено, .dsp1 фајл који се користи као улаз у CCS декодер.

У CCS декодеру је потребно имплементирати функцију **void decode(Int16* in, Int16* outL, Int16* outR, Int16 N, Uint16 BL, Uint16 BR)** , где је идеја да се у датој функцији имплементира горе наведени алгоритам декодовања. Параметри функције су:

in - `CompressedBuffer` - јединствени улазни канал;

outL - OutputBufferL - леви излазни канал;

outR - OutputBufferR - десни излазни канал;

N - AUDIO_IO_SIZE - МАКРО, дужине 128, представља дужину излазних канала;

BL - број бита за реконструкцију левог канала;

BR - број бита за реконструкцију десног канала.

2.2 Преузимање вредности канала

Супротно процесу учешљавања, у декодеру је потребно из јединственог улазног бафера, одбирке поделити на два канала. Имплементација овог дела декодера се налази у оквиру горепоменуте функције decode(), а њен изглед је следећи:

```
58      for(i = 0; i < 2*N; i++){
59          fft_bufferL[i] = in[2*i];
60      }
61
62      for(i = 0; i < 2*N; i++){
63          fft_bufferR[i] = in[2*i + 1];
64      }
```

Слика 7. Преузимање вредности канала

2.3 Реконструкција

То је инверзан процес квантизацији, где квантоване вредности враћамо на оне пре квантизације. У ову сврху је имплементирана функција **Int16 reconstructB(Int16 input, Uint16 B)**, која за сваки одбирак врши реконструкцију за задати број бита. Имплементација овог дела декодера се налази у оквиру горепоменуте функције decode(), а њен изглед је следећи:

```
24 Int16 reconstructB(Int16 input, Uint16 B)
25 {
26     Int16 output = input << (16-B);
27
28     return output;
29 }
30
31
```

Слика 8. Функција reconstruct()

```
92
93          //*************************************************************
94          //VRACANJE U OPSEG, REKONSTRUKCIJA
95
96          //Za LEVI kanal
97
98          for(i = 0; i < 2*N; i++){
99              fft_bufferL[i] = reconstructB(fft_bufferL[i], BL);
100         }
101
102         //Za DESNI kanal
103
104         for(i = 0; i < 2*N; i++){
105             fft_bufferR[i] = reconstructB(fft_bufferR[i], BR);
106         }
107     }
108     //*************************************************************
```

Слика 9. Део кода који врши реконструкцију

2.4 Прелазак(враћање) у времески домен(RIFFT)

Идеја је да се над обрађеним каналима fft_bufferL и fft_bufferR изврши инверзна Фуријеова трансформација, како би се они вратили у временски домен, те да се након тога примени метода преклопи и сабери, како би реконструкција била што боља. Након овог корака добијамо вредности излазних канала, које касније исписујемо у излазну датотеку. Имплементација овог дела декодера се налази у оквиру горепоменуте функције decode(), а њен изглед је следећи:

```
150     //*****************************************************************************
151     //INVERZNA FFT
152     //Za LEVI kanal
153
154     rifft(fft_bufferL, FFT_SIZE, NOSCALE);
155
156     //ovde treba breakpoint LEVI
157
158     for(i = 0; i < N; i++){
159         outL[i] = _smpy(fft_bufferL[i], window[i]) + _smpy(out_delayL[i], window[i + N]);
160         outL[i] = outL[i] * 4;
161         out_delayL[i] = fft_bufferL[i + N];
162     }
163
164     //Za DESNI kanal
165
166     rifft(fft_bufferR, FFT_SIZE, NOSCALE);
167
168     for(i = 0; i < N; i++){
169         outR[i] = _smpy(fft_bufferR[i], window[i]) + _smpy(out_delayR[i], window[i + N]);
170         outR[i] = outR[i] * 4;
171         out_delayR[i] = fft_bufferR[i + N];
172     }
173     //*****************************************************************************
174 }
```

Слика 10. Имплементација RIFFT

****

3.Прва контролна тачка

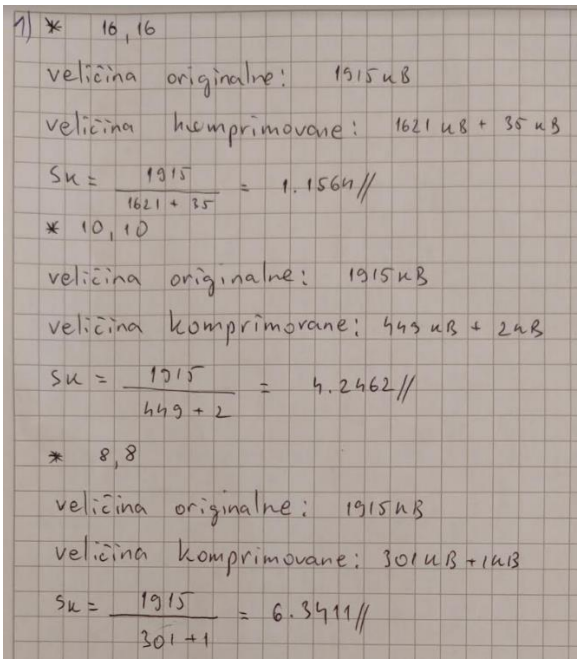
3.1 Први задатак

Идеја првог задатка је да се реализују енкодер и декодер, баш као што је то описано у поглављима: Увод, Енкодер, Декодер. Такође, након успешне реализације, потребно је испробати исти поступак за различите вредности улазних бајтова, те упоредити излазе на основу квалитета, као и компресоване фајлове на основу величине. Све то потребно је описати у текстуалном фајлу Zadatak1.txt, који је остављен уз решење.

```
ZADATAK1
*SK - Stepen kompresije
** Komentar o kvalitetu dekodovanog signala (subjektivni osećaj na osnovu slušanja)

Datoteka      | quantBL   |  quantBR |   MS   |  SK* |  Komentar*
-----------------------------------------------------------------------------------------------------------------------
stream1.wav   |    16     |    16    |    0   |1.1564|Dobijeni signal je gotovo identičan ulaznom signalu.
-----------------------------------------------------------------------------------------------------------------------
stream1.wav   |    10     |    10    |    0   |4.2462|Dobijeni signal nesto manjeg kvaliteta, ali i dalje dosta verodostojan.
-----------------------------------------------------------------------------------------------------------------------
stream1.wav   |    8      |     8    |    0   |6.3411|Dobijeni signal je vidno ostecen, ali i dalje prepoznatljiv.
-----------------------------------------------------------------------------------------------------------------------

Dodatni komentar: Jasno je da je najbolji odnos SK i kvaliteta zvuka u slucaju kada su BL i BR jednaki 10.
```

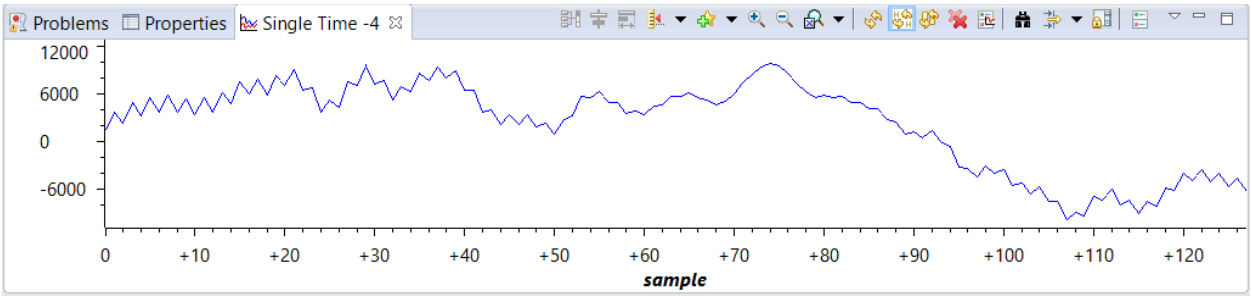
Слика 11. Излазни текстуални фајл из 1. задатка

Јасно је да смо имали три различита случаја: онај који користи 16, 10 и 8 битова за квантизацију и реконструкцију. У зависности од тих променљивих величина, мењају се и квалитет излазног сигнала, као и величина компресованог фајла.

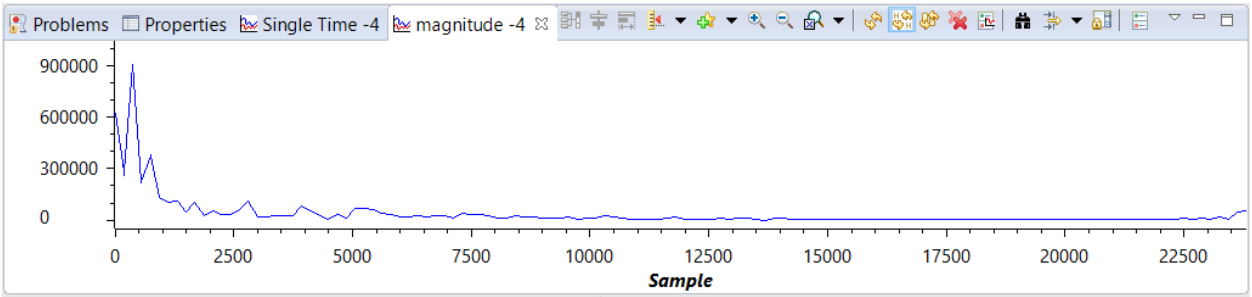


Слика 12. Начин рачунања степена компресије

За сваки пар улаза, било је потребно сачувати компримоване фајлове, речнике, излазне фајлове из декодера, те израчунати степен компресије. Такође је било потребно исцртати графике који показују стање fft_buffera у временском и фреквентном домену у различитим тренуцима. Коначно, потребно је било приказати излазни сигнал у временском и фреквентном домену.



Слика 13. излазни сигнал у временском домену(10 бита)



Слика 14. излазни сигнал у фреквентном домену(10 бита)

****

4. Друга контролна тачка

4.2 Други задатак

У другом задатку је циљ био да модификујемо већ постојећу структуру енкодера и декодера, те да убацимо још једну фазу обраде, која се зове нелинеарна обрада. Идеја је да се ова фаза извршава између фаза FFT и квантизације код енкодера, или између реконструкције и RIFFT за случај декодера. Нелинеарна обрада треба да над сваким одбирком канала fft_bufferL и fft_bufferR изврши следећу функцију:

$$X = sign(X) * \sqrt{X}$$,

тј. потребно је квадрирати сваки одбирак понаособ, притом водећи рачуна о његовом знаку.

Имплементација нелинеарне обраде извршена је са стране енкодера и декодера. За случај енкодера, она се налази у encode.c изворном фајлу, у функцији encode().

```
99    //*****************************************************
100   //NELINEARNA OBRADA
101   for(i = 0; i < 2*N; i++){
102       xL[i] = fft_bufferL[i];
103       xR[i] = fft_bufferR[i];
104   }
105
106   //ZNAKOVI
107   for(i = 0; i < 2*N; i++){
108       znakL[i] = sign(xL[i]);
109       znakR[i] = sign(xR[i]);
110   }
111
112   for(i = 0; i < 2*N; i++){
113       if(znakL[i] == -1){
114           xL[i] *= (-1);
115       }
116
117       if(znakR[i] == -1){
118           xR[i] *= (-1);
119       }
120   }
121
122
123   //sad su svi pozitivni
124
125   //KORENI
126   sqrt_16(xL, xL, FFT_SIZE);
127   sqrt_16(xR, xR, FFT_SIZE);
128
129   //Za LEVI kanal
130   for(i = 0; i < 2*N; i++){
131       xL[i] = znakL[i] * xL[i];
132       fft_bufferL[i] = xL[i];
133   }
134
135   //Za DESNI kanal
136   for(i = 0; i < 2*N; i++){
137       xR[i] = znakR[i] * xR[i];
138       fft_bufferR[i] = xR[i];
139   }
140   //*****************************************************
```

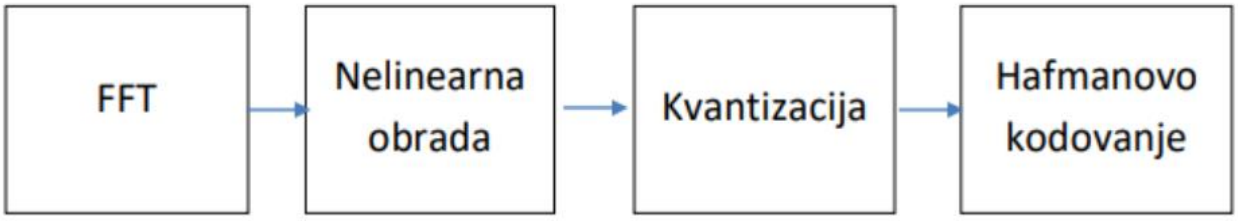
Слика 16. Нелинеарна обрада за енкодер

Са стране декодера, нелинеарна обрада ради инверзну операцију. Наиме, она сваки одбирак квадрира, притом водећи рачуна о знаку одбирка. Имплементација се налази у decode.c изворном фајлу, у функцији decode().

```
112     //*********************************************
113     //NELINEARNA OBRADA
114
115     //x = sign(x) * sqrt(x);
116     //x = sign(x) * pow(x, 2);
117
118     for(i = 0; i < 2*N; i++){
119         xL[i] = fft_bufferL[i];
120     }
121
122     for(i = 0; i < 2*N; i++){
123         xR[i] = fft_bufferR[i];
124     }
125
126     for(i = 0; i < 2*N; i++){
127         znakL[i] = sign(xL[i]);
128         znakR[i] = sign(xR[i]);
129     }
130
131     //Za LEVI kanal
132
133     for(i = 0; i < 2*N; i++){
134         xL[i] = _smpy(xL[i], xL[i]);
135         xL[i] = znakL[i] * xL[i];
136         fft_bufferL[i] = xL[i];
137     }
138
139     //Za DESNI kanal
140
141     for(i = 0; i < 2*N; i++){
142         xR[i] = _smpy(xR[i], xR[i]);
143         xR[i] = znakR[i] * xR[i];
144         fft_bufferR[i] = xR[i];
145     }
146     //*********************************************
```

Слика 17. Нелинеарна обрада за декодер

Након имплементације ове фазе обраде, структура енкодера изгледа овако:

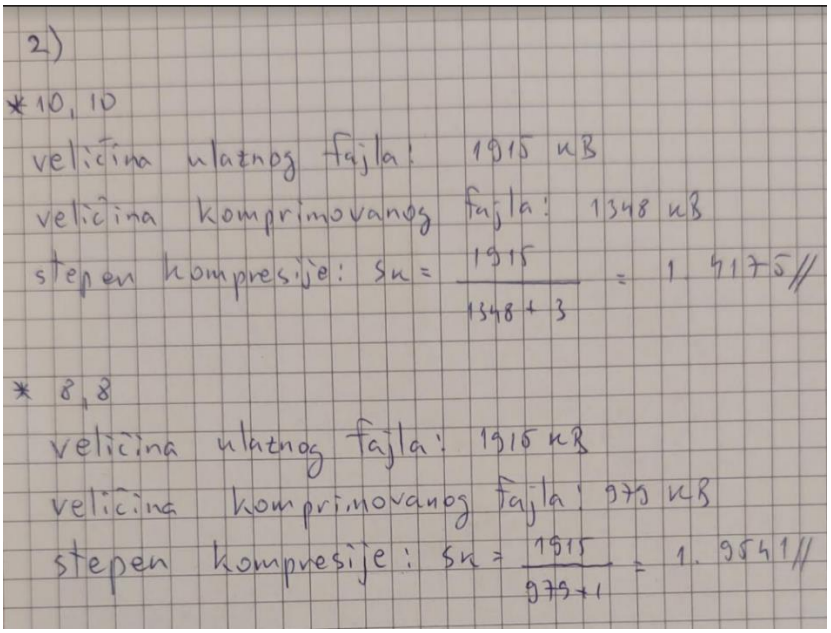


Слика 18. Структура модификованог енкодера

Као излаз из задатка је потребно уз већ стандардне .c датотеке приложити и компресовани фајл, речник и излазни фајл из декодера, као и попуњен текстуални фајл Zadatak2.txt.

```
ZADATAK2
*SK - Stepen kompresije
** Komentar o kvalitetu dekodovanog signala (subjektivni osećaj na osnovu slušanja)

Datoteka      | quantBL   |  quantBR |   MS   |  SK* |  Komentar*
-------------------------------------------------------------------------------------------------------------------------------------------
stream1.wav   |    10     |    10    |    0   |1.4175| Dobijen gotovo identican signal kao i u zadatku 1.
-------------------------------------------------------------------------------------------------------------------------------------------
stream1.wav   |    8      |     8    |    0   |1.9541|Neocekivano dobar zvuk, dosta bolji nego u zadatku 1., a slican kao za slucaj sa 10 bita.
-------------------------------------------------------------------------------------------------------------------------------------------

Dodatni komentar:
Interesantno je da je SK za ovaj zadatak dosta manji nego kada nije bilo nelinearne obrade.
Verovatno je to zbog nekih nepotrebnih stvari u kodu koje su napisane zbog intuitivnosti koda.
Takodje, jako interesantno je da smanjeni broj bita, sa 10 na 8 nije uticao na gubitak kvaliteta zvuka.
Medjutim, doslo je do smanjenja velicine komprimovanog fajla, sto je pozitivno.
```

Слика 19. Излазни текстуални фајл из 2. задатка



Слика 20. Начин рачунања степена компресије

****

4.3 Трећи задатак

У трећем задатку је било потребно имплементирати здружено кодовање, тзв. **joint coding**. То је један од најбољих начина компресије, који у потпуности задржава квалитет излазног фајла. Идеја је да се на основу улазних канала L и R формирају нови канали M и S, по следећој формули:

$$M(i) = \frac{InputBufferL(i) + inputBufferR(i)}{2}$$

$$S(i) = \frac{InputBufferL(i) - inputBufferR(i)}{2}$$

Наравно ово је потребно имплементирати у енкодеру, да бисмо касније применили инверзну функцију у декодеру и добили жељени излаз. Успех оваквог кодовања зависи од једнакости левог и десног канала. Наиме, ако су они једнаки, новоформирани канал S ће након прве модификације бити једнак нули, услед чега се у даљим фазама обраде ради само са M каналом. Међутим уколико почетни канали нису исти, након прве модификације S неће бити 0, те ће се у обради створити шум, што се код мене и десило.

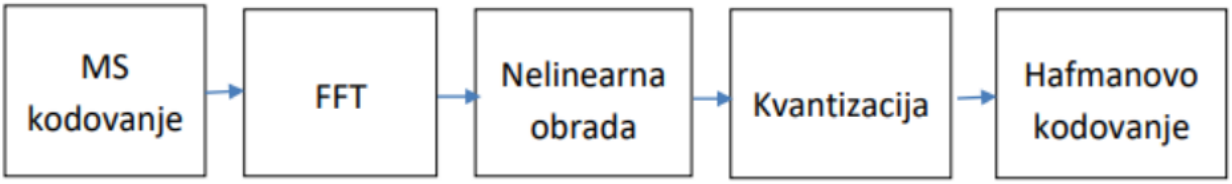
```
if(MS == 1){
    for(j = 0; j < AUDIO_IO_SIZE; j++)
    {
        M[j] = (InputBufferL[j] + InputBufferR[j])/2;
        S[j] = (InputBufferL[j] - InputBufferR[j])/2;
        InputBufferL[j] = M[j];
        InputBufferR[j] = S[j];
    }
}
```

Слика 21. Имплементација здруженог кодовања у енкодеру

```
if(MS == 1){
    for(j = 0; j < AUDIO_IO_SIZE; j++)
    {
        M[j] = OutputBufferL[j];
        S[j] = OutputBufferR[j];
        OutputBufferL[j] = (M[j] + S[j]);
        OutputBufferR[j] = (M[j] - S[j]);
    }
}
```

Слика 22. Имплементација здруженог кодовања у декодеру

Било је потребно имплементирати функционалност која ће у зависности од тога да ли то корисник хоће или не, покренути здружено кодовање. То је одрађено маневрисањем са флегом MS, који се ставља на 1 ако хоћемо здружено кодовање, или се оставља на иницијалној нули, ако не желимо исто. Након убацивања ове фазе обраде у енкодер, он графички изгледа овако:

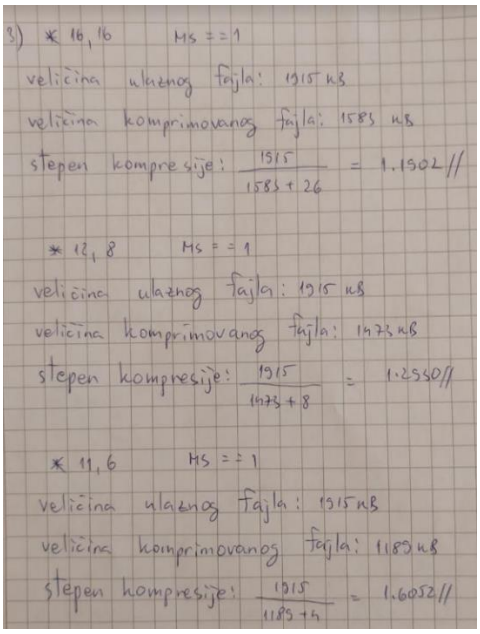


Слика 23. Структура модификованог енкодера

Излаз из задатка је као и у претходним случајевима компресовани фајл, речник и излазни фајл из декодера за сваки случај из табеле, као и табела сама по себи.

```
ZADATAK3
*SK - Stepen kompresije
** Komentar o kvalitetu dekodovanog signala (subjektivni osećaj na osnovu slušanja)

Datoteka      | quantBL   |  quantBR |   MS   |  SK* |  Komentar*
-------------------------------------------------------------------------------------------------------------------------------------------
stream1.wav   |    16     |    16    |    1   |1.1902|Signal u pozadini je isti kao i pocetni, ali ga nadmasuje sum koji se stvara u joint codingu.
-------------------------------------------------------------------------------------------------------------------------------------------
stream1.wav   |    12     |     8    |    1   |1.2930|Neprepoznatljiv signal.
-------------------------------------------------------------------------------------------------------------------------------------------
stream1.wav   |    11     |     6    |    1   |1.6052|Neprepoznatljiv signal.
-------------------------------------------------------------------------------------------------------------------------------------------

Napomena:

Ulazni signal za moj slucaj je bio "15.wav". Nakon dugo dekodovanja i nemogucnosti da nadjem gresku u kodu, ustanovio sam sledece:
Nakon ubacivanja signala u alat Audacity, jasno je da levi i desni kanal nisu isti.
To ce usloviti da nakon stvaranja novih kanala M i S i dalje obrade dodje do pojave suma u izlaznom signalu iz dekodera.
Naime, u joint codingu je ideja da kanal S nakon oduzimanja postane ceo popunjen nulama, sto se u mom slucaju ne desava.
Jasno je da ce razlike u kanalima dati vrednost razlicitu od 0.
To sam potvrdio ubacivanjem drugih ulaznih signala u moj kod(za "10.wav", "21.wav", "30.wav" sam dobijao ocekivan izlaz), gde sam dobijao ocekivane vrednosti.
Sa druge strane pojedini drugi ulazi daju rezultate slicne mojima. Kod svih ulaza koji daju dobre rezultate levi i desni kanal su skoro identicni, vazi i obrnuto.
```

Слика 24. Излазни текстуални фајл из 3. задатка



Слика 25. Начин рачунања степена компресије

4.4 Четврти задатак

У четвртом задатку је било потребно у зависности од бита BL и BR имплементирати механизам који би покретао специјални вид компресовања. Наиме, ако је један од бита једнак 0, нема смисла квантизовати и реконструисати сигнале са 0 бита, стога се уколико корисник унесе тај број бита, покреће специјалан вид компресије. Компресију је потребно урадити на следећи начин:

- Opseg 0 – 100 Hz – anulirati
- Opseg 100Hz – 512Hz – kvantizovati sa 8 bita
- Opseg 512Hz – 4096Hz – kvantizovati sa 12 bita
- Opseg 4096Hz – 14336Hz – kvantizovati sa 6 bita
- Sve iznad 14336Hz anulirati

,

Ово је имплементирано у оквиру енкодера и декодера. Код енкодера се реализација врши у оквиру квантизације на следећи начин:

```
144     //************************************************************
145     //KVANTIZACIJA
146
147     if(BL == 0 || BR == 0){
148         Int16 k1 = round((double)FFT_SIZE / 48000 * 100);
149         Int16 k2 = round((double)FFT_SIZE / 48000 * 512);
150         Int16 k3 = round((double)FFT_SIZE / 48000 * 4096);
151         Int16 k4 = round((double)FFT_SIZE / 48000 * 14336);
152
153         for(i = 0; i < 2*N; i++){
154             if(i < k1){
155                 fft_bufferL[i] = 0;
156                 fft_bufferR[i] = 0;
157             }else if(i < k2){
158                 fft_bufferL[i] = quantB(fft_bufferL[i], (Uint16)8);
159                 fft_bufferR[i] = quantB(fft_bufferR[i], (Uint16)8);
160             }else if(i < k3){
161                 fft_bufferL[i] = quantB(fft_bufferL[i], (Uint16)12);
162                 fft_bufferR[i] = quantB(fft_bufferR[i], (Uint16)12);
163             }else if(i < k4){
164                 fft_bufferL[i] = quantB(fft_bufferL[i], (Uint16)6);
165                 fft_bufferR[i] = quantB(fft_bufferR[i], (Uint16)6);
166             }else{
167                 fft_bufferL[i] = 0;
168                 fft_bufferR[i] = 0;
169             }
170         }
```

Слика 26. Измењена квантизација у енкодеру

Уколико ниједан од улазних бита није једнак нули, процес се одвија на начин као у 3. задатку.

Код декодера је извршена модификација реконструкције на следећи начин:

```
66      if(BL == 0 || BR == 0){
67          Int16 k1 = round((double)FFT_SIZE / 48000 * 100);
68          Int16 k2 = round((double)FFT_SIZE / 48000 * 512);
69          Int16 k3 = round((double)FFT_SIZE / 48000 * 4096);
70          Int16 k4 = round((double)FFT_SIZE / 48000 * 14336);
71
72          for(i = 0; i < 2*N; i++){
73              if(i < k1){
74                  fft_bufferL[i] = 0;
75                  fft_bufferR[i] = 0;
76              }
77              else if(i < k2){
78                  fft_bufferL[i] = reconstructB(fft_bufferL[i], (Uint16)8);
79                  fft_bufferR[i] = reconstructB(fft_bufferR[i], (Uint16)8);
80              }else if(i < k3){
81                  fft_bufferL[i] = reconstructB(fft_bufferL[i], (Uint16)12);
82                  fft_bufferR[i] = reconstructB(fft_bufferR[i], (Uint16)12);
83              }else if(i < k4){
84                  fft_bufferL[i] = reconstructB(fft_bufferL[i], (Uint16)6);
85                  fft_bufferR[i] = reconstructB(fft_bufferR[i], (Uint16)6);
86              }else{
87                  fft_bufferL[i] = 0;
88                  fft_bufferR[i] = 0;
89              }
90          }
```

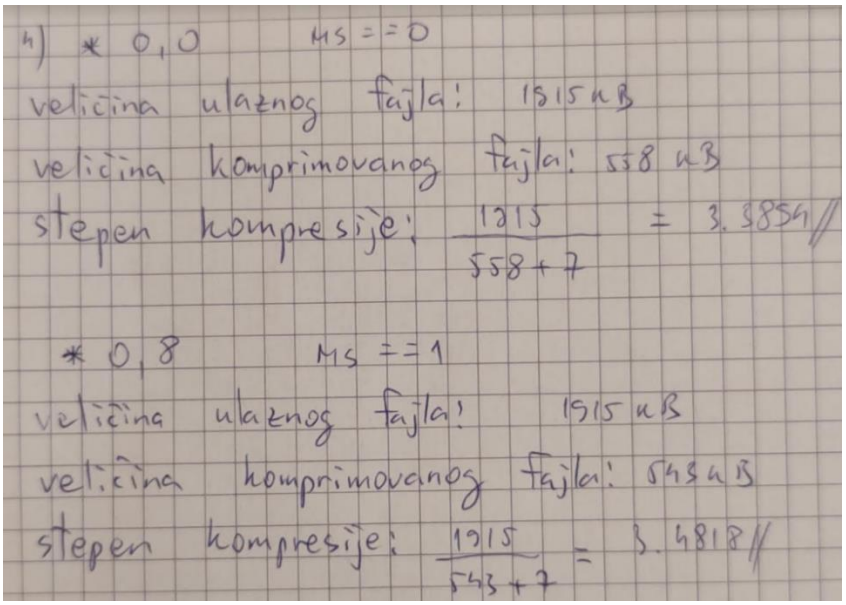
Слика 27. Измењена реконструкција у декодеру

Уколико ниједан од улазних бита није једнак нули, процес се одвија на начин као у 3. задатку.

Излаз из задатка је компресовани фајл, речник и излаз из декодера за сваки пар у табели, као и сама табела. Текстуални фајл Zadatak4.txt изгледа овако:

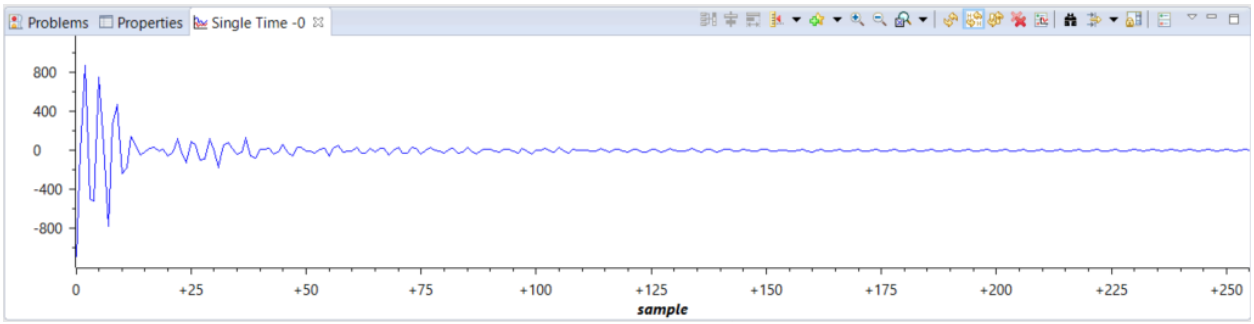
```
ZADATAK4
*SK - Stepen kompresije
** Komentar o kvalitetu dekodovanog signala (subjektivni osećaj na osnovu slušanja)

Datoteka      | quantBL   |  quantBR |   MS   |  SK* |  Komentar*
-------------------------------------------------------------------------------------------------------------------------
stream1.wav   |     0     |     0    |    0   |3.3894|Na desnoj kanalu signal je ocekivan, na levom postoji jak sum.
-------------------------------------------------------------------------------------------------------------------------
stream1.wav   |     0     |     8    |    1   |3.4818|Na oba kanala jak sum, sto je ista posledica kao i u trecem zadatku.
-------------------------------------------------------------------------------------------------------------------------

Dodatni komentar:
U oba slucaja se stvara sum. On je posledica odabiranja sa razlicitim brojem bita(8, 12, 6), sa jedne strane,
ili sa druge strane istog fenomena koji se desava u trecem zadatku, kada usled S != 0 -> sum.
Kada sam uvrstio u prvom slucaju isti broj bita(npr. 8, umesto 8, 12, 6), dobio sam cist signal na oba kanala.
Nakom uvrstanja redom bita: 8, 12, 6, u levom kanalu se stvara sum, a desni ostaje nepromenjen.
```

Слика 28. Излазни текстуални фајл из 3. задатка

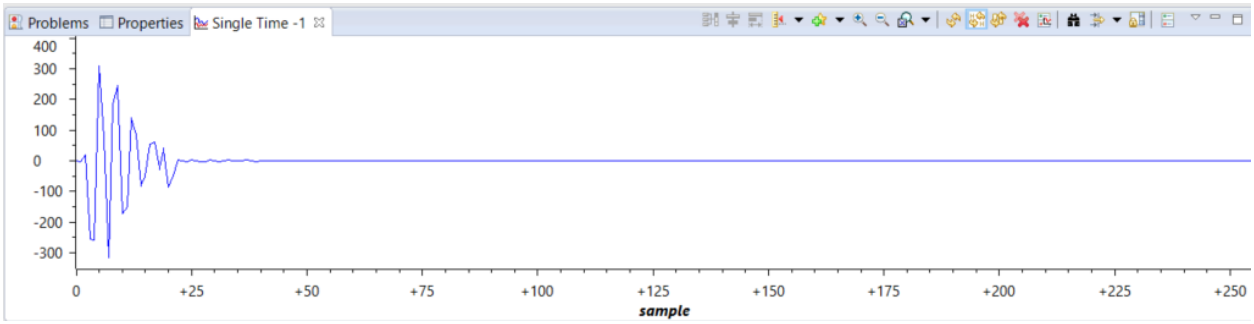


Слика 29. Начин рачунања степена компресије

Излаз је и садржај fft_buffera након Фуријеове трансформације, као и након квантизације за BL || BR == 0.



Слика 30. Садржај fft_bufferL након FFT



Слика 31. Садржај fft_bufferL након квантизације