

FPGA – based Embedded System Design

LAB #3

Group Members

Mohammad Taghizadeh Givari	810198373
Zeinab Saeedi	810198411
Amin Aroufzad	810198538

محتوا

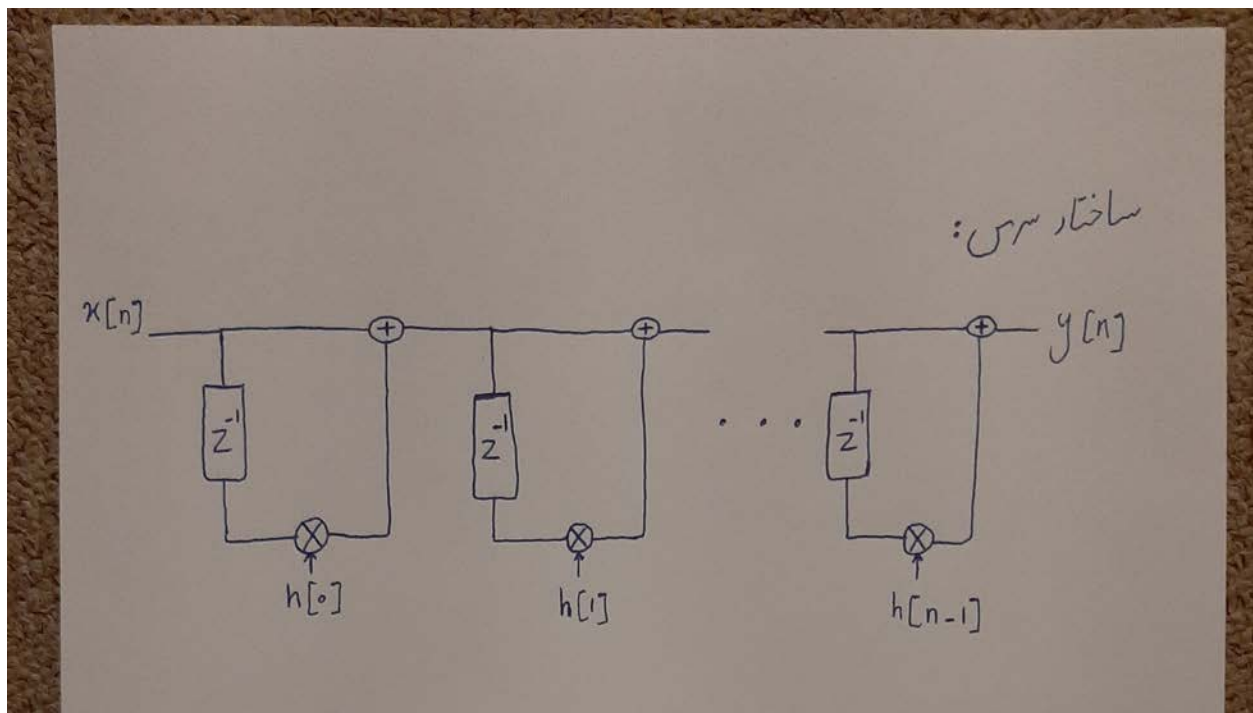
۱. طراحی سطح بالای فیلتر حذف نویز.....۲
۲. پیاده سازی نرم افزاری فیلتر حذف نویز.....۱۲
۳. پیاده سازی سخت افزاری فیلتر حذف نویز.....۱۷

طراحی سطح بالای فیلتر حذف نویز

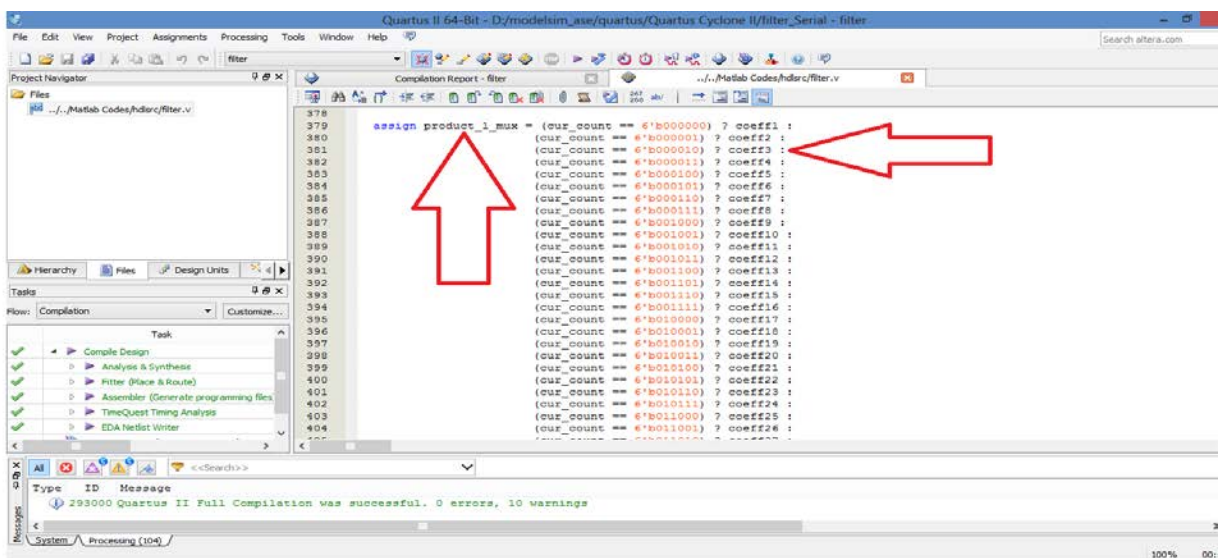
ساختار سری:

$$H(z) = (c_0 + c_1 z^{-1}) \times (c_2 + c_3 z^{-1}) \times \dots \times (c_{n-1} + c_n z^{-1})$$

شماتیک ساختار سری:



مالتی پلکسر (Multiplexer) ورودی:

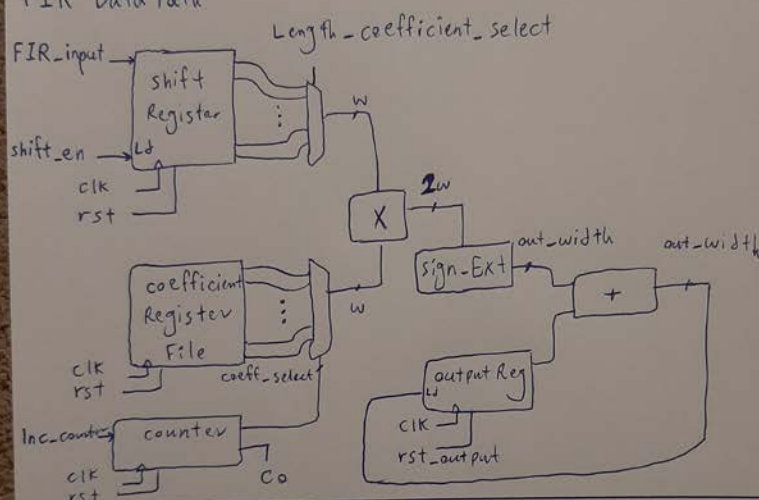


مسیر داده و کنترلر پیاده سازی ساختار سری تنها با یک جمع کننده و ضرب کننده:

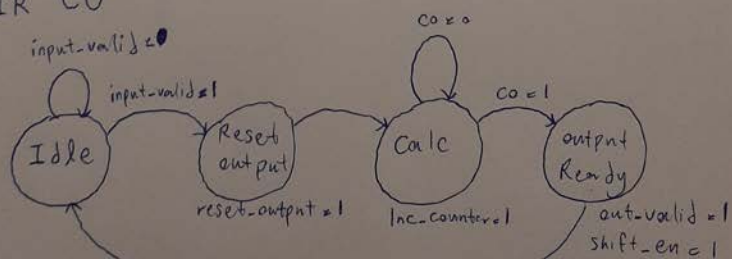
FIR Top module



FIR Data Path



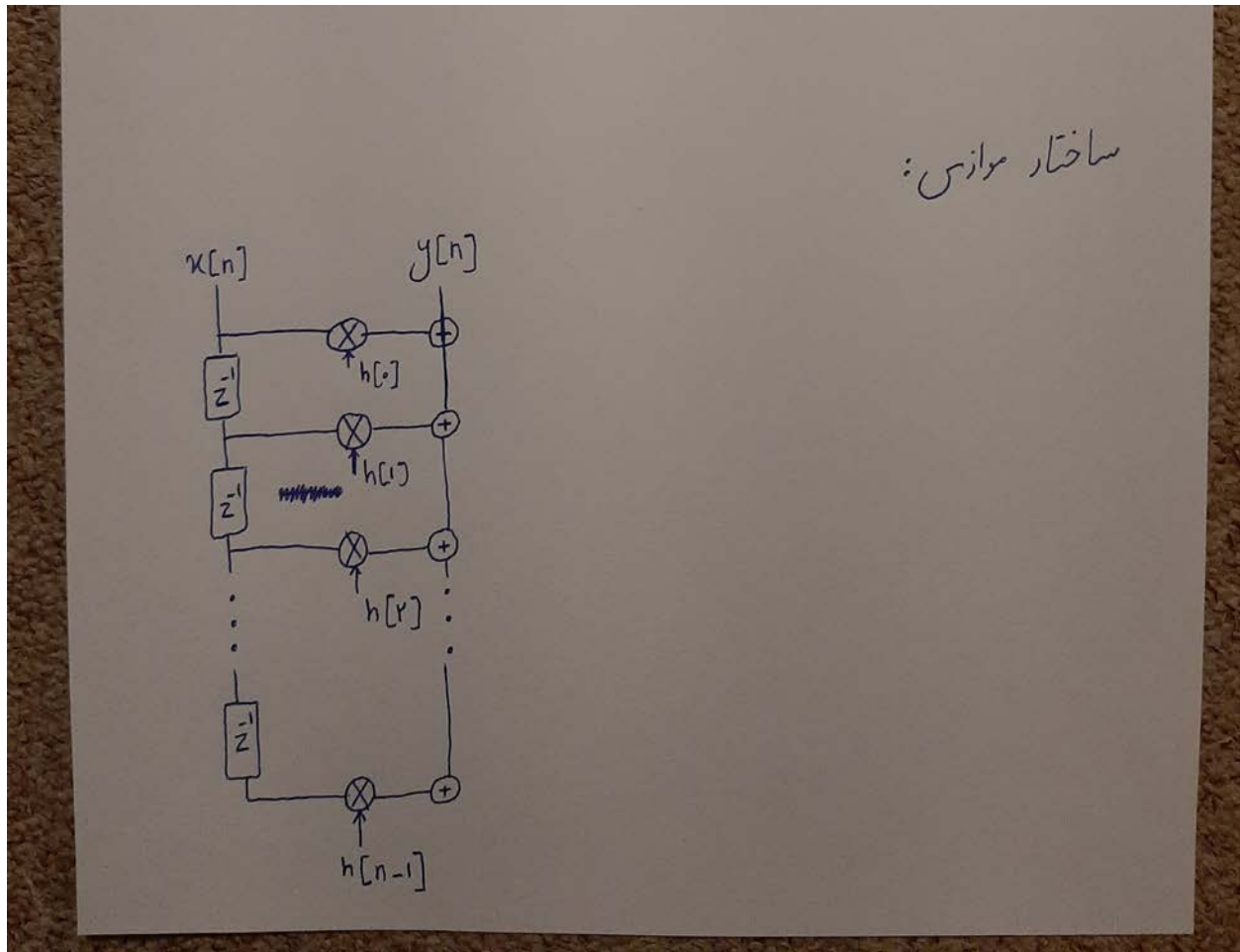
FIR CU



ساختار موازی:

$$H(z) = c_0 + c_1 z^{-1} + c_2 z^{-2} + \dots + c_n z^{-n}$$

شماتیک ساختار موازی:



مقایسه ساختار سری و موازی:

در ساختار سری فقط از یک جمع کننده و یک ضرب کننده استفاده شده است، اما در ساختار موازی از ۶۴ عدد ضرب کننده و یک جمع کننده استفاده شده است. در نتیجه:

- ۱- تعداد المان های منطقی مصرف شده در ساختار موازی به مراتب بیش تر از ساختار سری است
- ۲- تاخیر ساختار موازی به دلیل وجود تعداد زیادی ضرب کننده، به مراتب بیش تر از ساختار سری که فقط از یک ضرب کننده و جمع کننده استفاده کرده است، می باشد. در نتیجه فرکانس بیشینه مدار در حالت موازی به مراتب کمتر از فرکانس بیشینه در حالت سری است.

ساختار سری:

The screenshot shows the Quartus II 64-Bit software interface. The 'Fitter Summary' window is open, displaying the following data:

Item	Value	Percentage
Total logic elements	2,081	33,216 (6 %)
Total combinational functions	1,247	33,216 (4 %)
Dedicated logic registers	1,640	33,216 (5 %)
Total registers	1640	
Total pins	59	475 (12 %)
Total virtual pins	0	
Total memory bits	0	483,840 (0 %)
Embedded Multiplier 9-bit elements	4	70 (6 %)
Total PLLs	0	4 (0 %)

The status bar at the bottom indicates: 293000 Quartus II Full Compilation was successful. 0 errors, 10 warnings.

Quartus II 64-Bit - D:/modelsim_ase/quartus/Quartus Cyclone II/filter - filter

File Edit View Project Assignments Processing Tools Window Help

Project Navigator

Entity

Cyclone II: EP2C35F672C6

filter

ipm_mult0

Table of Contents

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	64.72 MHz	64.72 MHz	clk	

293000 Quartus II Full Compilation was successful. 0 errors, 10 warnings

System Processing (104)

100% 00:01

ساختار موازی:

Quartus II 64-Bit - D:/modelsim_ase/quartus/Quartus Cyclone II/filter - filter_Parallel

File Edit View Project Assignments Processing Tools Window Help

Project Navigator

Files

filter_Parallel

Table of Contents

Flow Summary

Flow Status	Successful - Sun Dec 18 12:25:28 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 S3 Web Edition
Revision Name	filter_Parallel
Top-level Entity Name	filter_Parallel
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	16,993 / 33,216 (51 %)
Total combinational functions	16,416 / 33,216 (49 %)
Dedicated logic registers	1,568 / 33,216 (5 %)
Total registers	1568
Total pins	59 / 475 (12 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	70 / 70 (100 %)
Total PLLs	0 / 4 (0 %)

293000 Quartus II Full Compilation was successful. 0 errors, 109 warnings

System Processing (399)

100% 00:01

The screenshot shows the Quartus II 64-Bit IDE interface. The 'Table of Contents' pane on the left highlights the 'Slow Model Fmax Summary' report. The main window displays the 'Fmax Summary' table, which lists the maximum frequency (Fmax) for each clock in the design. A red arrow points to the 'Fmax' column header.

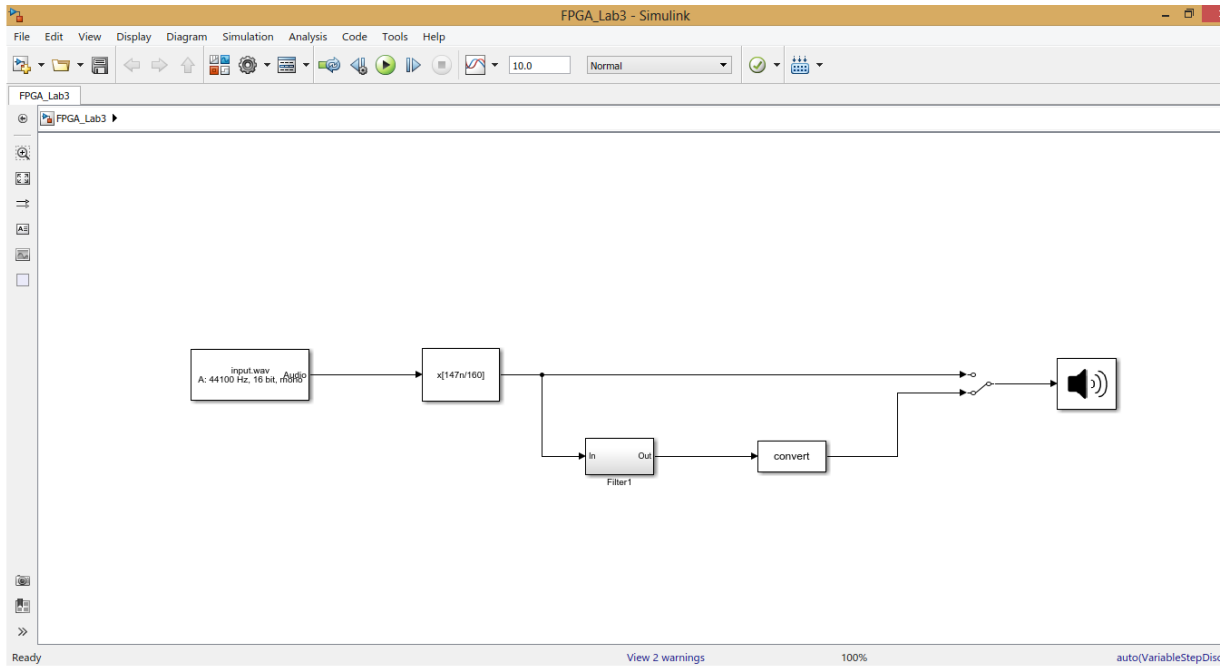
	Fmax	Restricted Fmax	Clock Name	Note
1	27.87 MHz	27.87 MHz	clk	

Below the table, a note states: "This panel reports FMAX for every clock in the design, regardless of the user-specified clock periods. FMAX is only computed for paths where the source and destination registers or ports are driven by the same clock. Paths of different clocks, including generated clocks, are ignored. For paths between a clock and its inversion, FMAX is computed as if the rising and falling edges are scaled along with FMAX, such that the duty cycle (in terms of a percentage) is maintained. Altera recommends that you always use clock constraints and other slack reports for sign-off analysis."

The 'Messages' pane at the bottom shows a successful compilation message: "293000 Quartus II Full Compilation was successful. 0 errors, 109 warnings".

همان طور که در تصاویر فوق میبینید، تعداد المان های منطقی مصرف شده در ساختار موازی به مراتب بیش تر از ساختار سری است و فرکانس بیشینه مدار در حالت موازی به مراتب کمتر از فرکانس بیشینه در حالت سری می باشد.

سیمولینک فیلتر طراحی شده توسط متلب:



تفاوت حالت Wrap با حالت Saturate:

حالت Wrap:

اگر سیگنال در اثر ضرب و جمع های متوالی در فیلتر، از دامنه دینامیکی قابل نمایش، خارج شود و overflow رخ دهد، سیگنال مجدد از صفر شروع به افزایش می کند:

$$1111...1 + 1 = 0000...0 \Rightarrow 0000...0 + 1 = 000...1 \Rightarrow \dots$$

به عبارتی سیگنال چرخیده (به اصطلاح Wrapped around شده) و دوباره از اول شروع به افزایش می کند.

حالت Saturate:

اگر سیگنال در اثر ضرب و جمع های متوالی در فیلتر، از دامنه دینامیکی قابل نمایش، خارج شود و overflow رخ دهد، سیگنال در بیشینه یا کمینه مقدار قابل نمایش، ثابت می ماند:

$$1111...1 + 1 = 1111...1, \quad 0000...0 - 1 = 000...0$$

به عبارتی سیگنال در مقادیر بیشینه و کمینه خود اشباع (Saturate) شده و دیگر تغییر نمی کند.

مزایا و معایب کوانتیزه کردن ضرایب فیلتر:

مزایا:

ضرایب فیلتر در حالت کلی آنالوگ هستند. برای اینکه FPGA که مداری دیجیتال است، بتواند بر روی این ضرایب پردازش انجام دهد، لازم است این ضرایب به صورت دیجیتال، قابلیت نمایش داشته باشند. به همین دلیل ضرایب را کوانتیزه میکنیم. در واقع مزیت کوانتیزه کردن ضرایب فیلتر آن است که با این کار، می توان ضرایب فیلتر را به صورت دیجیتال نمایش و بر روی آنان، پردازش های مختلف را انجام داد.

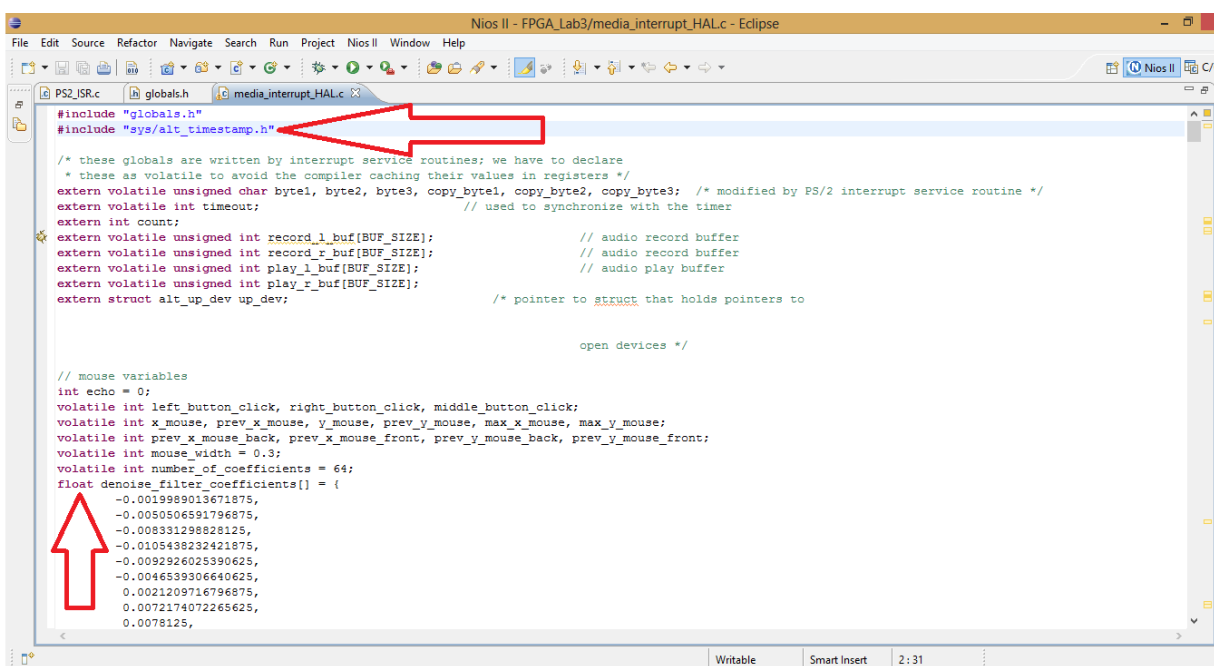
معایب:

عیب اصلی کوانتیزه کردن، آن است که با کوانتیزه و گرد کردن ضرایب، خطا در مقادیر ضرایب ایجاد می شود و در نتیجه اعداد کوانتیزه شده، متفاوت از ضرایب اصلی هستند که این امر موجب بروز خطا در محاسبات فیلتر می شود.

پیاده سازی نرم افزاری فیلتر حذف نویز

گام اول:

ابتدا کتابخانه مربوط به اندازه گیری زمان اجرا "sys/alt_timestamp.h" را include می کنیم تا بتوانیم از آن برای اندازه گیری زمان اجرا حذف نویز نرم افزاری، استفاده کنیم. سپس ضرایب فیلتر تولید شده توسط متلب را استخراج و مطابق تصویر زیر، به کد اضافه می کنیم تا از این ضرایب برای حذف نویز به صورت نرم افزاری استفاده کنیم:



```
#include "globals.h"
#include "sys/alt_timestamp.h"

/* these globals are written by interrupt service routines; we have to declare
 * these as volatile to avoid the compiler caching their values in registers */
extern volatile unsigned char byte1, byte2, byte3, copy_byte1, copy_byte2, copy_byte3; /* modified by PS/2 interrupt service routine */
extern volatile int timeout; /* used to synchronize with the timer */
extern int count;
extern volatile unsigned int record_l_buf[BUF_SIZE]; /* audio record buffer */
extern volatile unsigned int record_r_buf[BUF_SIZE]; /* audio record buffer */
extern volatile unsigned int play_l_buf[BUF_SIZE]; /* audio play buffer */
extern volatile unsigned int play_r_buf[BUF_SIZE];
extern struct alt_up_dev up_dev; /* pointer to struct that holds pointers to
                                open devices */

// mouse variables
int echo = 0;
volatile int left_button_click, right_button_click, middle_button_click;
volatile int x_mouse, prev_x_mouse, y_mouse, prev_y_mouse, max_x_mouse, max_y_mouse;
volatile int prev_x_mouse_back, prev_x_mouse_front, prev_y_mouse_back, prev_y_mouse_front;
volatile int mouse_width = 0.3;
volatile int number_of_coefficients = 64;
float denoise_filter_coefficients[] = {
    -0.0019989013671875,
    -0.0050506591796875,
    -0.008331298828125,
    -0.0105438292421875,
    -0.0092926025390625,
    -0.0046539306640625,
    0.0021209716796875,
    0.0072174072265625,
    0.0078125,
```

گام دوم:

برای حذف نویز یک تابع به نام `denoise_the_noisy_sound` تعریف می کنیم:

```
void denoise_the_noisy_sound()
{
    int n;
    for(n = 0; n < BUF_SIZE; n++)
    {
        //printf("%d\n", n);
        float result = 0;
        int k;
        for(k = 0; k <= n && k < number_of_coefficients; k++)
        {
            float record = (float)(record_r_buf[n - k] >> 8);
            float temp = denoise_filter_coefficients[k] * (record / (2 ^ 23));
            result = result + temp;
        }
        play_l_buf[n] = (int)(result * (2 ^ 30));
        play_r_buf[n] = (int)(result * (2 ^ 30));
        //printf("%d\n", play_l_buf[n]);
    }
}

void echo_maker(unsigned int l_buf[], unsigned int r_buf[], unsigned int* echo_l_buf, unsigned int* echo_r_buf) {
    int i;
    for (i = 0; i < BUF_SIZE; ++i) {
        if (i >= ECHO_INDEX1) {
            if (i >= ECHO_INDEX2) {
                echo_l_buf[i] = l_buf[i] >> 1 + l_buf[i - ECHO_INDEX2] >> 2;
                echo_r_buf[i] = r_buf[i] >> 1 + r_buf[i - ECHO_INDEX2] >> 2;
            }
            else {
                echo_l_buf[i] = l_buf[i] >> 1 + l_buf[i - ECHO_INDEX1] >> 2;
                echo_r_buf[i] = r_buf[i] >> 1 + r_buf[i - ECHO_INDEX1] >> 2;
            }
        }
        else {
            echo_l_buf[i] = l_buf[i];
            echo_r_buf[i] = r_buf[i];
        }
    }
}
```

مطابق تصویر فوق، برای حذف نویز به صورت نرم افزاری در یک حلقه ضرایب را در سیگنال ضبط شده مطابق فرمول

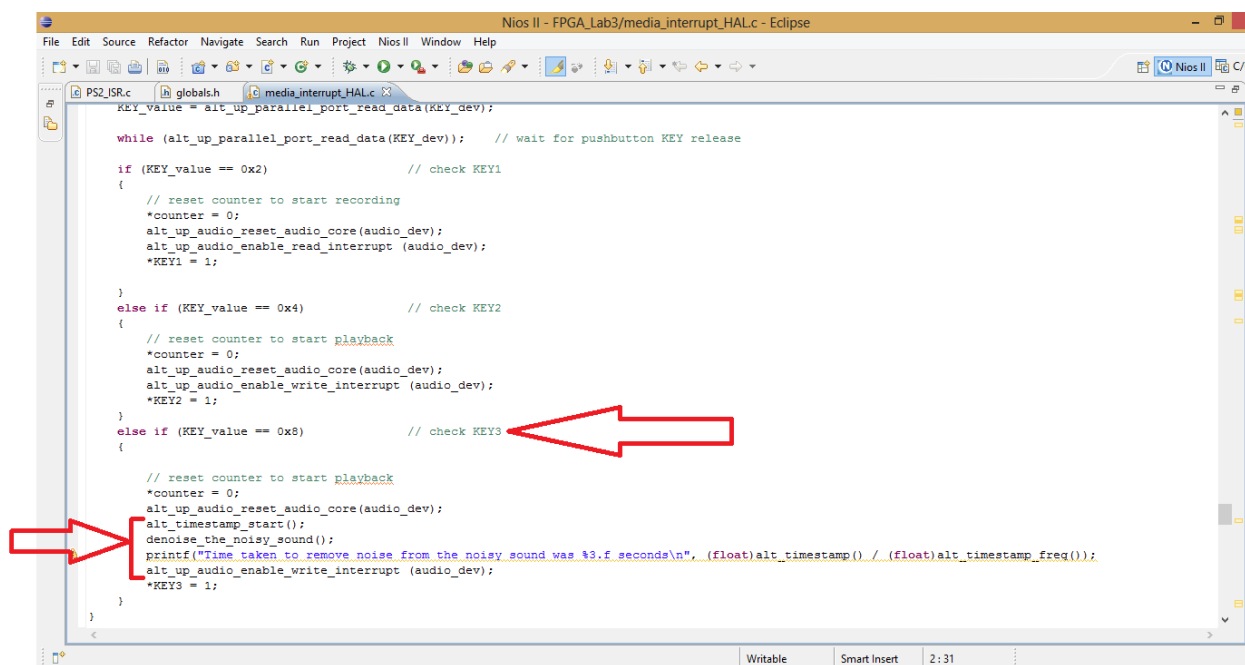
$$y[n] = \sum_{k=0}^m b[k].x[n - k]$$

ضرب کردیم. از جایی که داده های ضبط شده در ۲۴ بیت با ارزش `record_r_buf` هستند. با ۸ بیت شیفต์ دادن `record_r_buf` به سمت راست، داده های ضبط شده در متغیر `record_r_buf` باقی می ماند. ضرایب تولید شده توسط متلب در صورتی درست عمل می کنند که ورودی فیلتر ۲۴ بیت با ۲۳ بیت اعشار، و خروجی فیلتر ۳۲ بیت با ۳۰ بیت اعشار باشند. به همین دلیل داده ضبط شده را تقسیم بر ۲ به توان ۲۳ کردیم تا داده ضبط شده به صورت ۲۴ بیت با ۲۳ بیت اعشار شود و

در آخر هم خروجی که به صورت ۳۲ بیت با ۳۰ بیت اعشار هست را ضرب در ۲ به توان ۳۰ کردیم تا تمام ۳۲ بیت خروجی از قسمت اعشار خارج و در قسمت صحیح قرار گیرند و بتوان در فرمت integer تمام بیت های خروجی را مشاهده کرد.

در آخر با یک حلقه تو درتو، در هر بار اجرا حلقه داخلی، یک خروجی ساخته می شود و بعد از اجرا شدن حلقه بیرونی تمام خروجی ها ساخته خواهد شد.

گام سوم:



```
KEY_value = alt_up_parallel_port_read_data(KEY_dev);

while (alt_up_parallel_port_read_data(KEY_dev)); // wait for pushbutton KEY release

if (KEY_value == 0x2) // check KEY1
{
    // reset counter to start recording
    *counter = 0;
    alt_up_audio_reset_audio_core(audio_dev);
    alt_up_audio_enable_read_interrupt (audio_dev);
    *KEY1 = 1;
}

else if (KEY_value == 0x4) // check KEY2
{
    // reset counter to start playback
    *counter = 0;
    alt_up_audio_reset_audio_core(audio_dev);
    alt_up_audio_enable_write_interrupt (audio_dev);
    *KEY2 = 1;
}

else if (KEY_value == 0x8) // check KEY3
{
    // reset counter to start playback
    *counter = 0;
    alt_up_audio_reset_audio_core(audio_dev);
    alt_timestamp_start();
    denoise_the_noisy_sound();
    printf("Time taken to remove noise from the noisy sound was %3.f seconds\n", (float)alt_timestamp() / (float)alt_timestamp_freq());
    alt_up_audio_enable_write_interrupt (audio_dev);
    *KEY3 = 1;
}
}
```

مطابق تصویر فوق، در تابع `check_keys`، چک می کنیم که اگر کلید ۳ فشرده شد:

- `audio_core` را ریست می کنیم.

- با استفاده از `alt_timestamp_start`، محاسبه زمان اجرا را شروع می کنیم.

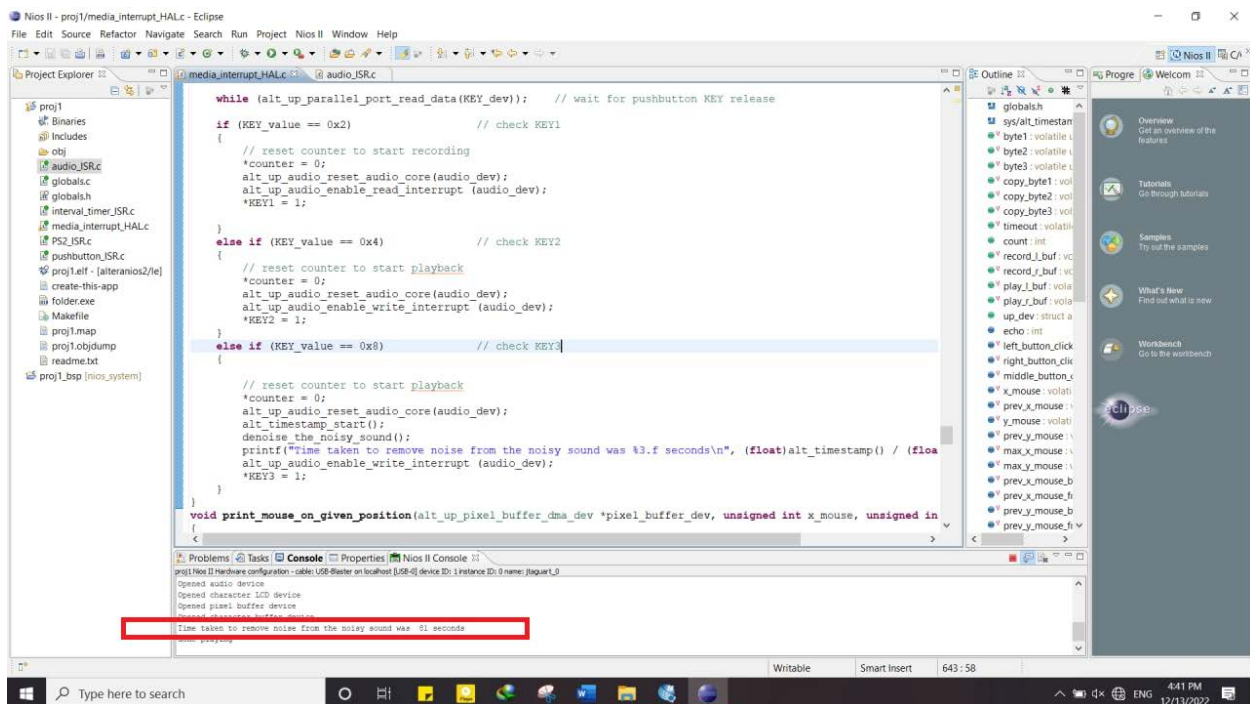
- با `denoise_the_noisy_sound`، صوت ضبط شده در `record_buf` را حذف نویز می کنیم.

- در آخر با تقسیم تعداد تیک های زده شده (`alt_timestamp`) بر تعداد تیک هایی که در واحد زمان

زده می شود (`alt_timestamp_freq`)، زمان اجرا تابع حذف نویز محاسبه شده و بر روی کنسول

چاپ می شود.

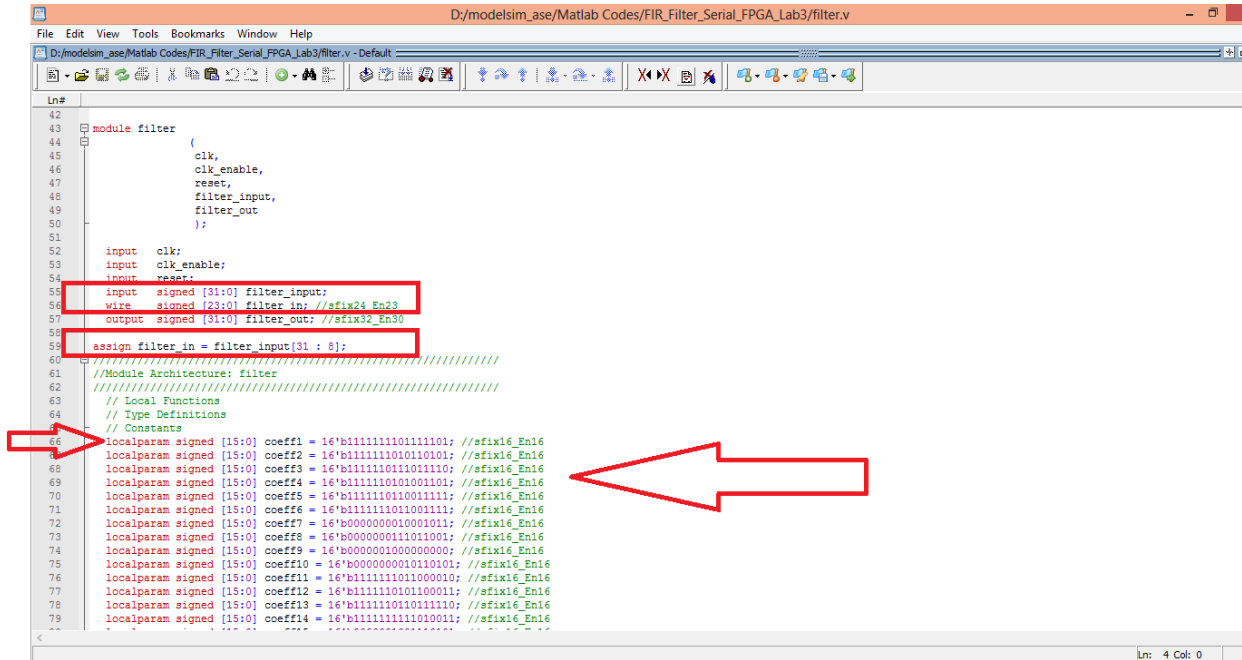
خروجی پیاده سازی نرم افزاری فیلتر حذف نویز بر روی برد DE2 Media Computer



همان طور که در تصویر فوق میبینید، پس از ۸۱ ثانیه، صوت ضبط شده به صورت نرم افزاری حذف نویز شده است.

پیاده سازی سخت افزاری حذف نویز

گام دوم (تغییرات کد Verilog ساخته شده توسط متلب):

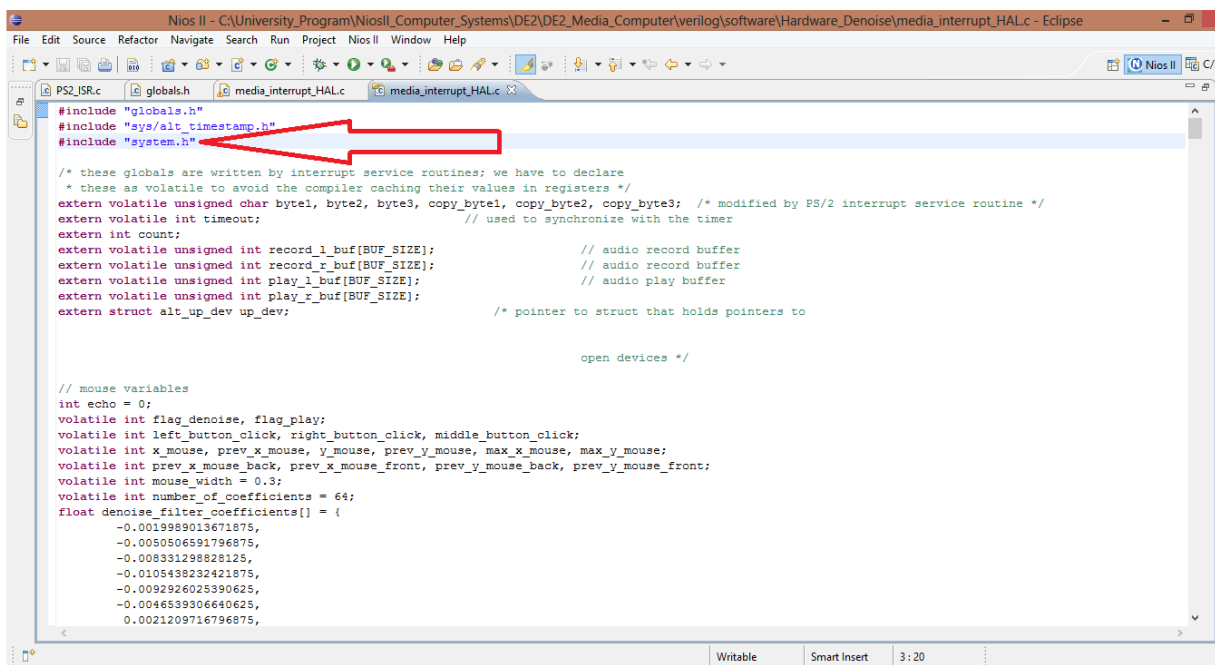


```
42
43 module filter
44 (
45     clk,
46     clk_enable,
47     reset,
48     filter_input,
49     filter_out
50 );
51
52 input clk;
53 input clk_enable;
54 input reset;
55 input signed [31:0] filter_input;
56 wire signed [23:0] filter_in; //sfix24_En23
57 output signed [31:0] filter_out; //sfix32_En30
58
59 assign filter_in = filter_input[31:8];
60
61 //Module Architecture: filter
62 //Local Functions
63 // Type Definitions
64 // Constants
65 localparam signed [15:0] coeff1 = 16'b1111111101111101; //sfix16_En16
66 localparam signed [15:0] coeff2 = 16'b1111111101010101; //sfix16_En16
67 localparam signed [15:0] coeff3 = 16'b1111111101101110; //sfix16_En16
68 localparam signed [15:0] coeff4 = 16'b1111111101010110; //sfix16_En16
69 localparam signed [15:0] coeff5 = 16'b1111111101001111; //sfix16_En16
70 localparam signed [15:0] coeff6 = 16'b1111111101001111; //sfix16_En16
71 localparam signed [15:0] coeff7 = 16'b0000000010001011; //sfix16_En16
72 localparam signed [15:0] coeff8 = 16'b0000000011011001; //sfix16_En16
73 localparam signed [15:0] coeff9 = 16'b0000000100000000; //sfix16_En16
74 localparam signed [15:0] coeff10 = 16'b0000000010110101; //sfix16_En16
75 localparam signed [15:0] coeff11 = 16'b1111111101100010; //sfix16_En16
76 localparam signed [15:0] coeff12 = 16'b1111111101010001; //sfix16_En16
77 localparam signed [15:0] coeff13 = 16'b1111111101011110; //sfix16_En16
78 localparam signed [15:0] coeff14 = 16'b111111110100011; //sfix16_En16
```

مطابق تصویر فوق، از جایی که ورودی فیلتر حذف نویز، ۳۲ بیتی است ولی فقط ۲۴ بیت با ارزش آن در محاسبات استفاده می شود، یک input به اسم filter_input تعریف کردیم. سپس یک wire به اسم filter_in تعریف کرده و آن را با ۲۴ بیت با ارزش filter_input ([31:8]) مقداردهی کردیم. در آخر متغیر هایی که به صورت parameter تعریف شده بودند را به localparam تغییر دادیم تا به صورت محلی تعریف شوند و دیگر از بیرون فیلتر و با ورودی مقداردهی نشوند. (به طور مثال ضرایب فیلتر مقادیری هستند که به صورت پیش فرض درون فیلتر تعریف می شوند و نیازی به مقدار دهی آنان از بیرون فیلتر و به صورت ورودی نیست.)

گام سوم:

ابتدا "system.h" را include می کنیم تا بتوانیم از ماکروساخته شده برای دستور اختصاصی مربوط به فیلتر حذف نویز استفاده کنیم:



```
#include "globals.h"
#include "sys/alt_timestamp.h"
#include "system.h"

/* these globals are written by interrupt service routines; we have to declare
 * these as volatile to avoid the compiler caching their values in registers */
extern volatile unsigned char byte1, byte2, byte3, copy_byte1, copy_byte2, copy_byte3; /* modified by PS/2 interrupt service routine */
extern volatile int timeout; /* used to synchronize with the timer */
extern int count;
extern volatile unsigned int record_l_buf[BUF_SIZE]; /* audio record buffer */
extern volatile unsigned int record_r_buf[BUF_SIZE]; /* audio record buffer */
extern volatile unsigned int play_l_buf[BUF_SIZE]; /* audio play buffer */
extern volatile unsigned int play_r_buf[BUF_SIZE];
extern struct alt_up_dev up_dev; /* pointer to struct that holds pointers to

                                open devices */

// mouse variables
int echo = 0;
volatile int flag_denoise, flag_play;
volatile int left_button_click, right_button_click, middle_button_click;
volatile int x_mouse, prev_x_mouse, y_mouse, prev_y_mouse, max_x_mouse, max_y_mouse;
volatile int prev_x_mouse_back, prev_x_mouse_front, prev_y_mouse_back, prev_y_mouse_front;
volatile int mouse_width = 0.3;
volatile int number_of_coefficients = 64;
float denoise_filter_coefficients[] = {
    -0.0019989013671875,
    -0.0050506591796875,
    -0.008331298828125,
    -0.0105498232421875,
    -0.0092926025390625,
    -0.0046539306640625,
    0.0021209716796875,
```

```
alt_up_audio_reset_audio_core(audio_dev);
alt_up_audio_enable_read_interrupt (audio_dev);
*KEY1 = 1;

}
else if (KEY_value == 0x4)          // check KEY2
{
    // reset counter to start playback
    *counter = 0;
    alt_up_audio_reset_audio_core(audio_dev);
    flag_play = 1;
    alt_up_audio_enable_write_interrupt (audio_dev);
}
else if (KEY_value == 0x8)          // check KEY3
{
    // reset counter to start playback
    *counter = 0;
    alt_up_audio_reset_audio_core(audio_dev);
    alt_timestamp_start();

    int j = 0;
    int k = 0;
    for(j = 0; j < BUF_SIZE; j++){
        play_r_buf[j] = ALT_CI_FIR_FILTER_0(record_r_buf[j]);
    }
    for(k = 0; k < BUF_SIZE; k++){
        play_l_buf[k] = ALT_CI_FIR_FILTER_0(record_l_buf[k]);
    }
    printf("Time taken to remove noise from the noisy sound was %3.f seconds\n", (float)alt_timestamp() / (float)alt_timestamp_freq());
    flag_denoise = 1;
    alt_up_audio_enable_write_interrupt (audio_dev);
}
```

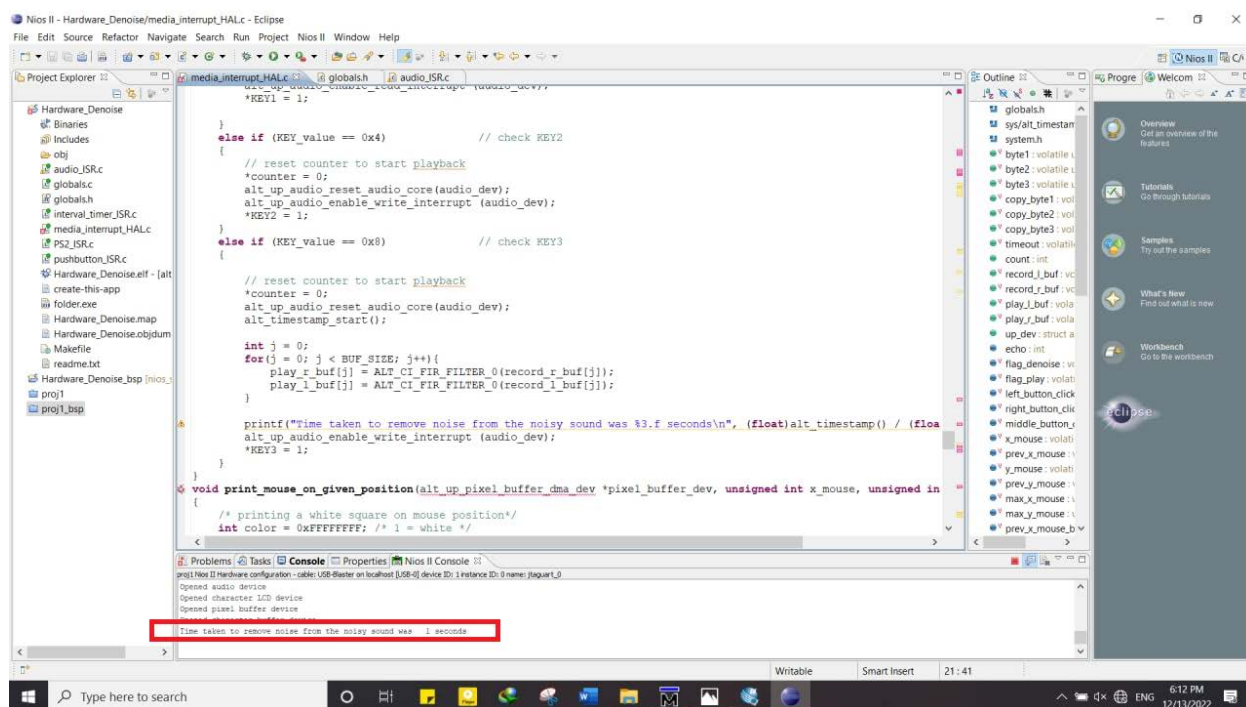
مطابق تصویر فوق، در یک حلقه با دادن داده ضبط شده (record_buf) به تابع مربوطه به دستور اختصاصی (ALT_CI_FIR_FILTER_0)، به ترتیب خروجی ها ساخته شده و در بافر مربوط به پخش صدا (play_buf) ذخیره می شوند. اینکار را یک بار برای بافر سمت راست (record_r_buf) و یک بار برای بافر سمت چپ (record_l_buf) انجام می دهیم.

سپس با تقسیم تعداد تیک های زده شده (alt_timestamp)، بر تعداد تیک هایی که در واحد زمان زده می شود (alt_timestamp_freq) زمان اجرا تابع حذف نویز محاسبه شده و بر روی کنسول چاپ می شود.

نکته:

اگر دو تا حلقه را یکی کنیم اتفاقی که میفتد آن است که ورودی فیلتر حذف نویز، در واحد زمان دوبار تکرار می شود در نتیجه در حوزه فرکانس طیف ورودی (صوت ضبط شده) بر ۲ تقسیم می شود و فرکانس نویز نیز بر ۲ تقسیم می شود. در آخر فرکانس نویز با تقسیم بر ۲ شدن و کوچک تر شدن، در محدوده عبوری فیلتر قرار گرفته و از فیلتر عبور می کند. در نتیجه صدا فیلتر شده همچنان دارای نویز خواهد بود. به همین دلیل دو حلقه را یکی نکردیم و جداگانه نوشتیم.

خروجی پیاده سازی سخت افزاری حذف نویز بر روی برد DE2 Media Computer



همان طور که در تصویر فوق میبینید، پس از ۱ ثانیه، صوت ضبط شده به صورت سخت افزاری حذف نویز شده است که این نسبت به ۸۱ ثانیه مربوط به حذف نویز به صورت نرم افزاری بسیار سریع تر است.

مقایسه میزان استفاده از منابع FPGA در پیاده سازی سخت افزاری و نرم افزاری فیلتر حذف نویز

میزان استفاده از منابع FPGA در پیاده سازی سخت افزاری فیلتر حذف نویز:

Table of Contents

Table of Contents	Flow Summary
Flow Summary	Successful - Tue Dec 20 22:31:24 2022
Flow Settings	Quartus II 64-Bit Version 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Flow Non-Default Global Settings	Revision Name DE2_Media_Computer
Flow Elapsed Time	Top-level Entity Name DE2_Media_Computer
Flow OS Summary	Family Cyclone II
Flow Log	Device EP2C35F672C6
Analysis & Synthesis	Timing Models Final
Fitter	Total logic elements 20,012 / 33,216 (60 %)
Flow Messages	Total combinational functions 17,210 / 33,216 (52 %)
Flow Suppressed Messages	Dedicated logic registers 13,275 / 33,216 (40 %)
Assembler	Total registers 13766
TimeQuest Timing Analyzer	Total pins 416 / 475 (88 %)
	Total virtual pins 0
	Total memory bits 170,297 / 483,840 (35 %)
	Embedded Multiplier 9-bit elements 25 / 70 (36 %)
	Total PLLs 2 / 4 (50 %)

293000 Quartus II Full Compilation was successful. 0 errors, 184 warnings

میزان استفاده از منابع FPGA در پیاده سازی نرم افزاری فیلتر حذف نویز:

Table of Contents

Table of Contents	Flow Summary
Flow Summary	Successful - Tue Dec 20 22:17:35 2022
Flow Settings	Quartus II 64-Bit Version 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Flow Non-Default Global Settings	Revision Name DE2_Media_Computer
Flow Elapsed Time	Top-level Entity Name DE2_Media_Computer
Flow OS Summary	Family Cyclone II
Flow Log	Device EP2C35F672C6
Analysis & Synthesis	Timing Models Final
Fitter	Total logic elements 17,860 / 33,216 (54 %)
Flow Messages	Total combinational functions 15,917 / 33,216 (48 %)
Flow Suppressed Messages	Dedicated logic registers 11,628 / 33,216 (35 %)
Assembler	Total registers 12119
TimeQuest Timing Analyzer	Total pins 416 / 475 (88 %)
	Total virtual pins 0
	Total memory bits 170,297 / 483,840 (35 %)
	Embedded Multiplier 9-bit elements 21 / 70 (30 %)
	Total PLLs 2 / 4 (50 %)

293000 Quartus II Full Compilation was successful. 0 errors, 181 warnings

همان طور که در تصاویر فوق میبینید، میزان استفاده از منابع FPGA در حالت پیاده سازی سخت افزاری نسبت به حالت پیاده سازی نرم افزاری بیش تر است که علت آن استفاده از منابع FPGA برای پیاده سازی سخت افزار مربوط به دستور اختصاصی (فیلتر حذف نویز به صورت سخت افزاری) می باشد.