به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

طراحی سیستم های نهفته مبتنی بر FPGA

آزمایش سوم

نام و نام خانوادگی	مهسا راستی – سعید شکوفا- محمد مهدی معینی منش
شماره دانشجویی	810198393 – 810198418 -810198475
تاریخ ارسال گزارش	1401.09.29

فهرست

1. طراحی سطح بالای فیلتر FIR و سنتز آن به کدRTL 5
مراحل ساخت فیلتر FIR در متلب 5
تفاوت حالت Saturate با حالت Wrap در زمان تنظیم مدOverflow 6
مزایا و معایب کوانتیزه کردن فیلتر 7
انتقال فیلتر به شبیه ساز متلب 7
فیلتر سریال 8
فیلتر موازی 9
عملکرد و مقایسه حالت موازی با حالت سری 11
2. اضافه کردن دستور اختصاصی به پردازنده یNios II 12
گام 1: اضافه کردن دستور اختصاصی به صورت نرم افزاری 12
گام 2: طراحی سخت افزاری فیلتر (ایجاد custom instruction جدید) 17
گام 3: استفاده از custom instruction تولید شده 20
سوال: نیازمندی های تعریف یک دستورMulticycle 22

شکل‌ها

شکل 1-1. طراحی فیلتر درجه 63 5
شکل 1-2. اندازه ضرایب 5
شکل 1-3. حالت Saturate 6
شکل 1-4. حالت Wrap 6
شکل 1-5. ساخت simulink متلب 7
شکل 1-6. فیلتر درست شده در simulink 7
شکل 1-7. ساختار سریال 8
شکل 1-8. فرکانس کاری حالت سریال 8
شکل 1-9. نمای زوم شده حالت سریال 8
شکل 1-10. نتایج سنتز حالت سریال 9
شکل 1-11. ساختار موازی 9
شکل 1-12. فرکانس کاری حالت موازی 10
شکل 1-13. نمای زوم شده حالت موازی 10
شکل 1-14. نتایج سنتز حالت موازی 10
شکل 2-1. تابع make_denoise 12
شکل 2-2. فیلتر طراحی شده 12
شکل 1-3. ضرایب coef 13
شکل 1-4. محاسبه زمان اجرای فیلتر 14
شکل 1-5. تنظیمات timestamp 15
شکل 1-6. تنظیمات timestamp 15
شکل 1-7. تنظیمات timestamp 15
شکل 1-8. زمان محاسبه شده حالت نرم افزاری 16
شکل 1-9. ادیت کد وریلاگ فیلتر 17
شکل 1-10. Component type 18
شکل 1-11. Files 18
شکل 1-12. Signals 19
شکل 1-13. interfaces 19

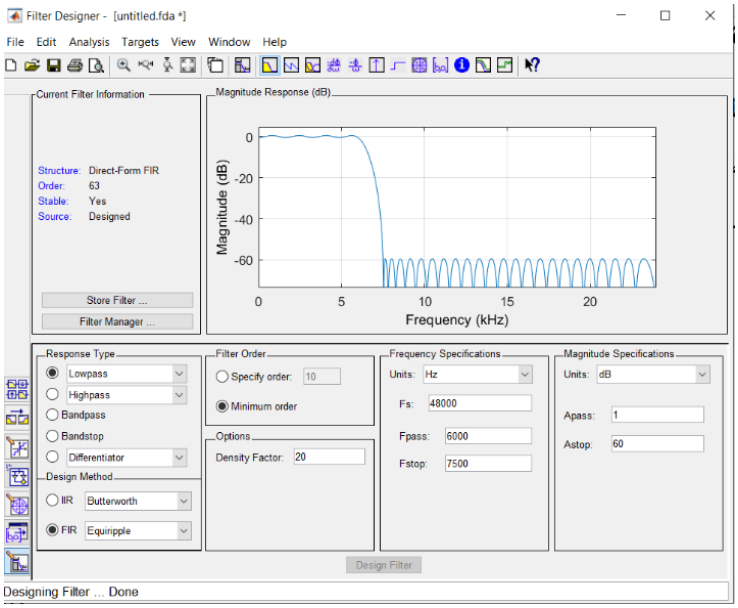
شکل 1-14. system contents 20
شکل 1-15. نتایج سنتز قبل از اضافه کردن custom instruction 20
شکل 1-16. نتایج سنتز بعد از اضافه کردن custom instruction 20
شکل 1-17. New Macro 21
شکل 1-18. دینویز کردن با ماکروی تولید شده 21
شکل 1-19. زمان اجرای فیلتر 22

جدول

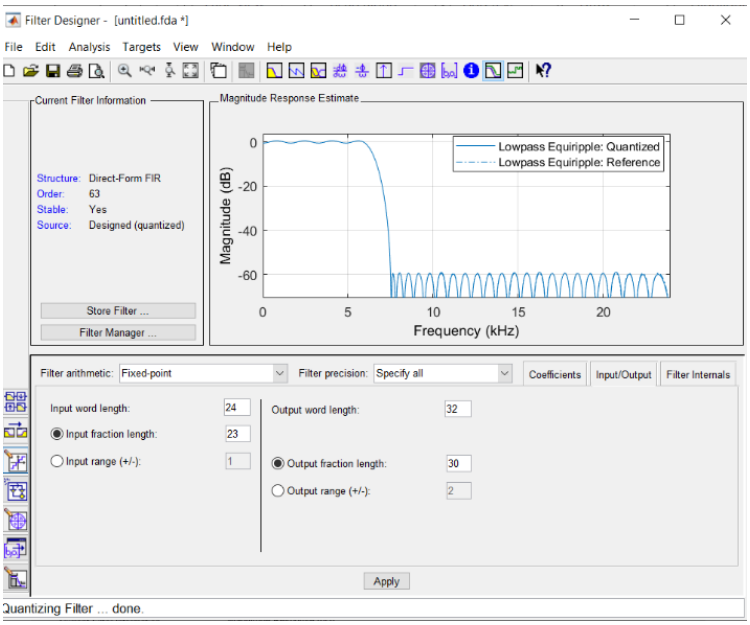
جدول 1- مقایسه میزان منابع 21

1. طراحی سطح بالای فیلتر FIR و سنتز آن به کد RTL

مراحل ساخت فیلتر FIR در متلب



شکل 1-1. طراحی فیلتر درجه 63

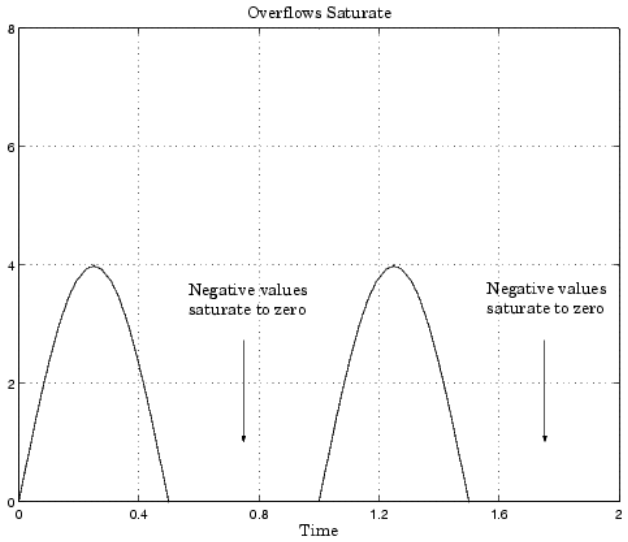


شکل 1-2. اندازه ضرایب

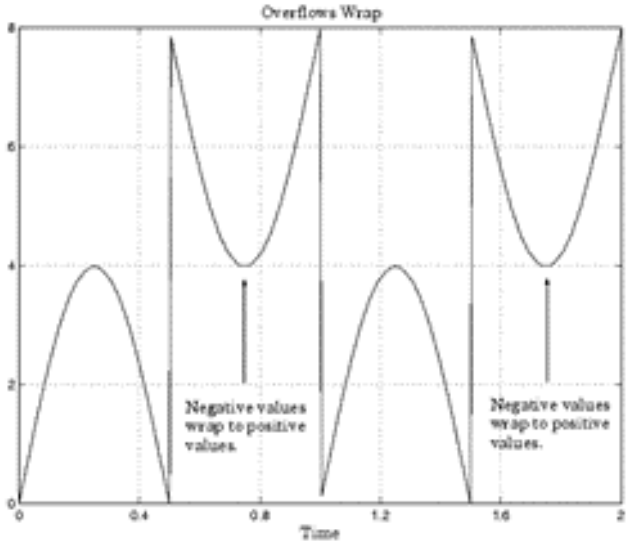
اندازه ضرایب طبق صورت گزارش (24 بیت به همراه 23 بیت اعشار برای ورودی و 32 بیت به همراه 30 بیت اعشار برای خروجی) تنظیم شده است.

تفاوت حالت Saturate با حالت Wrap در زمان تنظیم مد Overflow

پردازنده ها سیاست های متفاوتی برای برخورد با overflow دارند. حالت saturate اعداد منفی را به صفر (در واقع کوچک ترین عددی که میتواند نشان دهد) نظیر میکند و حالت wrap اعداد منفی را به اعداد مثبت به نحوی نظیر میکند که با اعداد مثبت اشتباه گرفته نشوند.



شکل 1-3. حالت Saturate

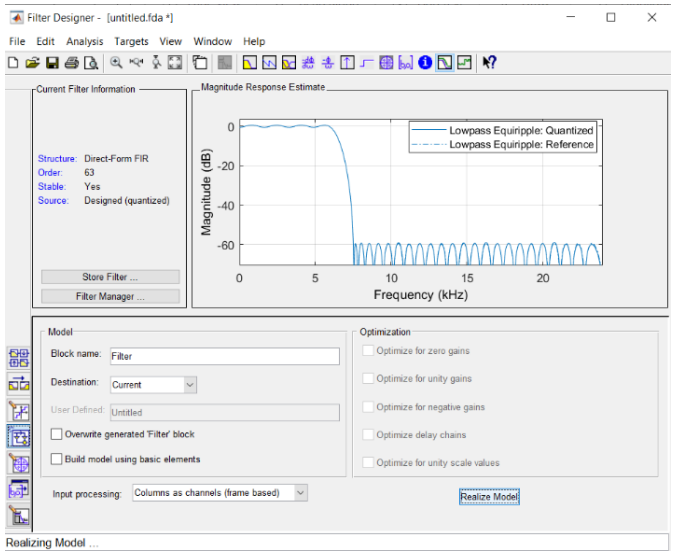


شکل 1-4. حالت Wrap

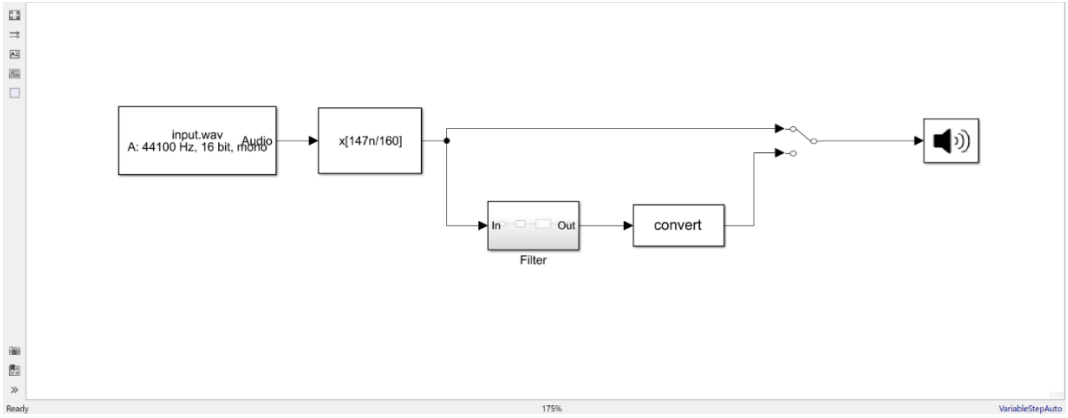
مزایا و معایب کوانتیزه کردن فیلتر

کوانتیزه کردن ضرایب این عیب را دارد که هنگامی که کوانتیزه می شوند با یک خطایی کوانتیزه می شوند. در واقع چون عدد ما 23 بیت اعشار دارد دقت ما به اندازه همان 23 بیت اعشار می شود و شامل خطا می شود. همچنین کوانتیزه کردن سبب می شود که سیگنال آنالوگ به دیجیتال تبدیل شود که خود خطایی را به همراه دارد. مزیت آن این است که اگر چه خطایی را به محاسبات اضافه میکند ولی این خطا حداقل مقدار ممکن بوده و سبب می شود که دقت ما در بالاترین حد ممکن باشد. در واقع چون ضرایب ثابت هستند تضعیف سیگنال به کمترین مقدار می رسد.

انتقال فیلتر به شبیه ساز متلب



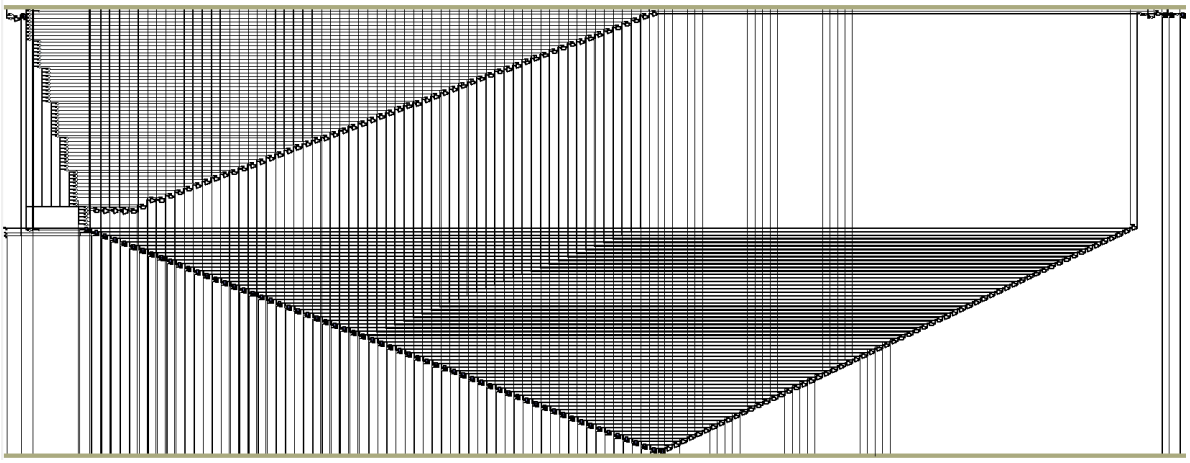
شکل 1-5. ساخت simulink متلب



شکل 1-6. فیلتر درست شده در simulink

فیلتر در متلب تست شد و مشاهده شد که به خوبی می تواند نویز را بگیرد و صدای شفاف را پخش کند.

فیلتر سریال

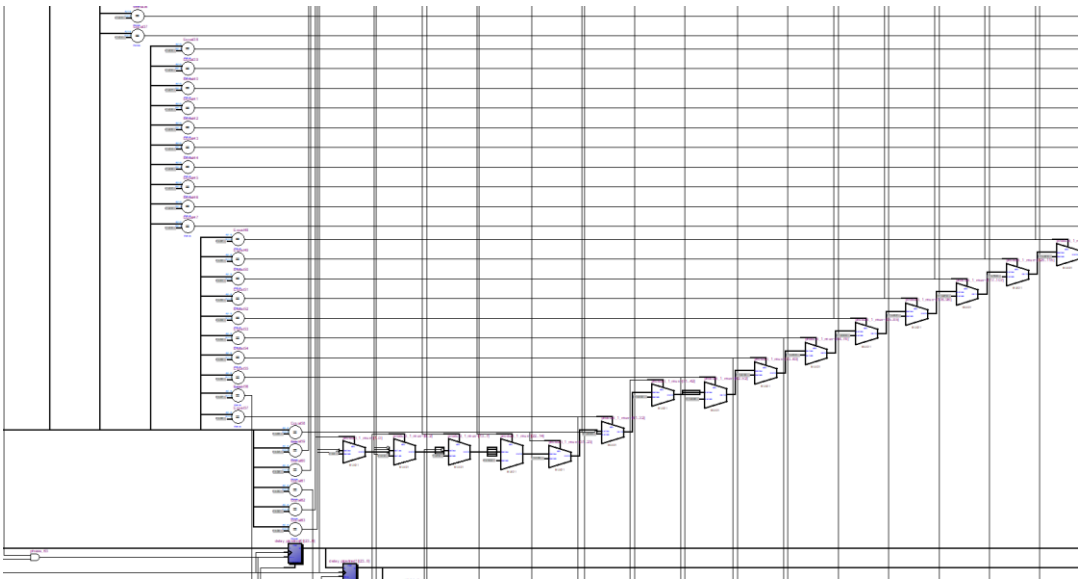


شکل 1-7. ساختار سریال

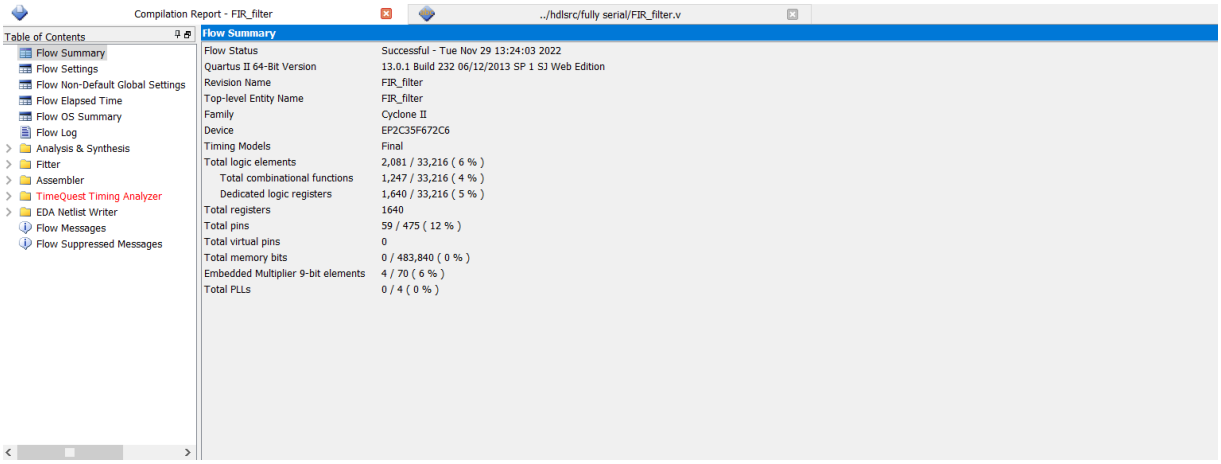
Slow Model Fmax Summary

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

شکل 1-8. فرکانس کاری حالت سریال



شکل 1-9. نمای زوم شده حالت سریال



شکل 1-10. نتایج سنتز حالت سریال

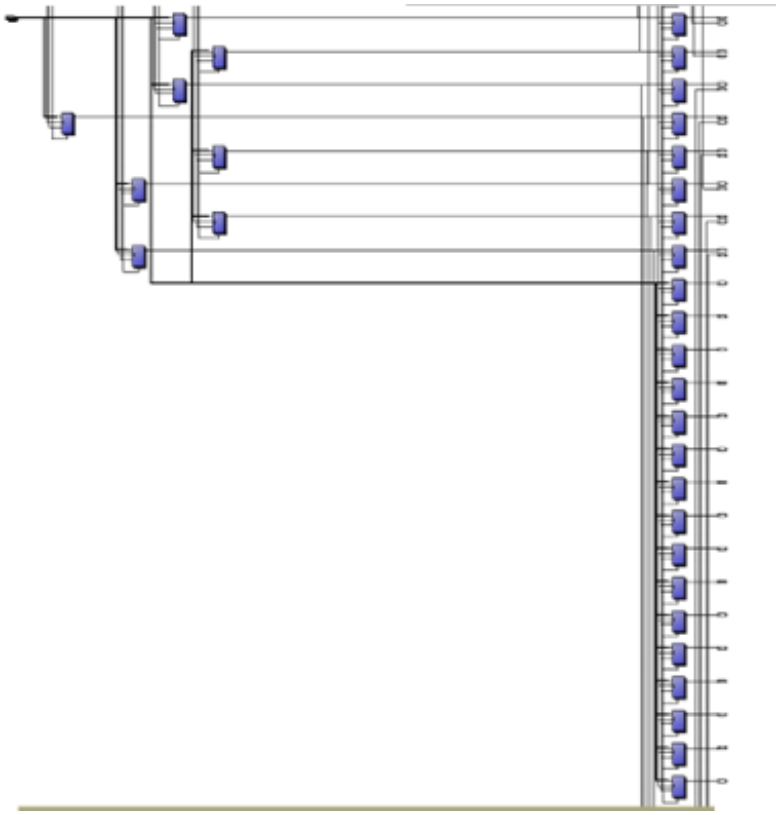
فیلتر موازی

شکل 1-11. ساختار موازی

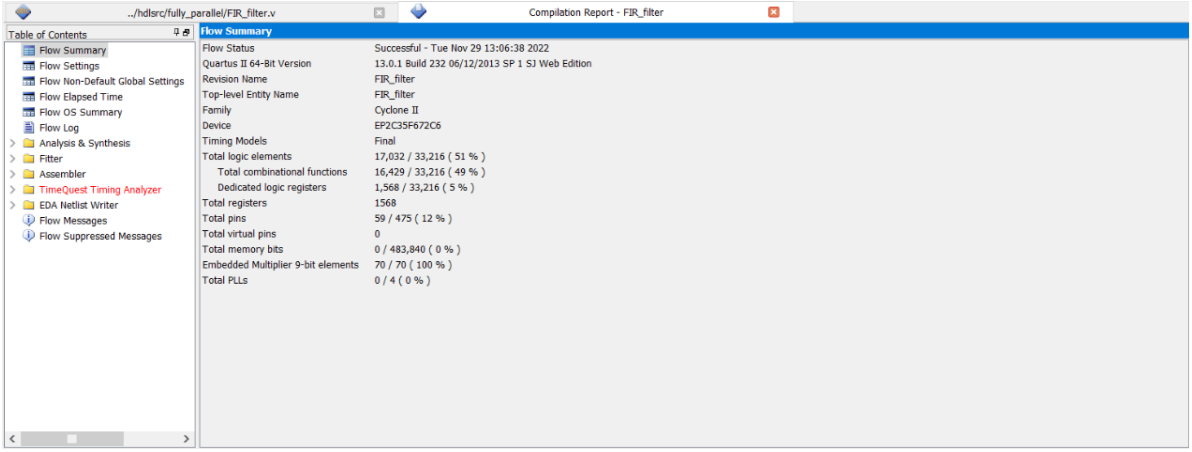
Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	25.34 MHz	25.34 MHz	clk	

شکل 1-12. فرکانس کاری حالت موازی



شکل 1-13. نمای زوم شده حالت موازی



شکل 1-14. نتایج سنتز حالت موازی

عملکرد و مقایسه حالت موازی با حالت سری

در حالت سری یک ضرب کننده و یک جمع کننده داریم و 64 تا رجیستر حالت تعریف شده است. در واقع با آمدن هر ورودی باید 64 تا کلاک صبر کرد تا خروجی آماده شود و این 64 تا کلاک با همان 64 تا رجیستر حالت که تعریف شده اند مشخص می شود. همچنین 64 تا رجیستر برای ذخیره 64 تا مقدار اخیر ورودی هستند در نظر گرفته شده است که در ضرایب ضرب و تقسیم شوند و خروجی را تولید کنند. در واقع در هر کدام از این 64 تا حالت با یکی از ورودی ها ضرب شده و با مقدار قبلی (مقدار اولیه صفر است) جمع می شوند.

در حالت موازی دیگر خبری از رجیستر حالت نیست. چرا که تمامی ورودی ها به طور همزمان با ضرایب ضرب می شوند و سپس با هم جمع می شوند یعنی 64 تا ضرب کننده داریم (به همراه 64 جمع کننده) و خروجی بعد از یک کلاک تولید می شود و لازم نیست که 64 تا کلاک صبر کرد. البته در این حالت critical path بیشتر بوده و سبب می شود که فرکانس کاری کمتر از حالت قبل شود همچنین 64 تا رجیستر برای نگه داشتن 64 ورودی اخیر فیلتر در نظر گرفته شده است.

همچنین با توجه به نتایج سنتز مشاهده می شود که میزان استفاده از منابع FPGA در حالت سریال بسیار کمتر از حالت موازی بوده. چرا که فقط یک ضرب کننده و یک جمع کننده داریم؛ ولی باید 64 تا کلاک برای تولید خروجی صبر کرد. در حالت موازی میزان استفاده از منابع زیاد بوده ولی خروجی در یک کلاک تولید می شود. هر چند که فرکانس کاری آن پایین تر است.

2. اضافه کردن دستور اختصاصی به پردازنده ی Nios II

گام 1: اضافه کردن دستور اختصاصی به صورت نرم افزاری

```
void make_Denoise(){
    int i;
    for (i = 0; i < BUF_SIZE; i++){
        l_buf_Denoise[i] = filter((double)l_buf[i]);
    }
    for (i = 0; i < 64; i++){
        mem[i] = 0;
    }
    for (i = 0; i < BUF_SIZE; i++){
        r_buf_Denoise[i] = filter((double)r_buf[i]);
    }
    for (i = 0; i < 64; i++){
        mem[i] = 0;
    }
}
```

شکل 2-1. تابع make_denoise

همانطور که در تصویر بالا مشخص است، دو حلقه for جداگانه برای بافر چپ و راست برای پخش استریو صدا استفاده شده است.

تابع make_denoise را جایگزین تابع make_echo کرده و نام متغیرهای مربوط به echo را به denoise تغییر میدهیم.

```
extern float mem[64];

unsigned int filter(unsigned int in){
    in = in >> 8;
    float float_in = (float) in / 8388608;

    int i;
    for (i=63;i>0;i--){
        mem[i]=mem[i-1];
    }
    mem[0]=in;
    double res=0;
    for(i=0;i<64;i++){
        res+=(double)mem[i]*(double)coef[i];
    }
    unsigned int result=(unsigned int)(res*1073741824);
    return result;
}
```

شکل 2-2. فیلتر طراحی شده

```
#include <stdio.h>

const float coef[64]={
-0.0019989013671875,
-0.0050506591796875,
-0.008331298828125 ,
-0.0105438232421875,
-0.0092926025390625,
-0.0046539306640625,
 0.0021209716796875,
 0.0072174072265625,
 0.0078125         ,
 0.0027618408203125,
-0.004852294921875 ,
-0.0102081298828125,
-0.008819580078125 ,
-0.0006866455078125,
 0.0095977783203125,
 0.0146331787109375,
 0.009735107421875 ,
-0.0036163330078125,
-0.0170440673828125,
-0.0205535888671875,
-0.009063720703125 ,
 0.01220703125     ,
 0.0296478271484375,
 0.028717041015625 ,
 0.00469970703125  ,
-0.031494140625    ,
-0.056121826171875 ,
-0.0446319580078125,
 0.013763427734375 ,
 0.106964111328125 ,
 0.2028656005859375,
 0.2635040283203125,
 0.2635040283203125,
 0.2028656005859375,
 0.106964111328125 ,
 0.013763427734375 ,
-0.0446319580078125,
-0.056121826171875 ,
-0.031494140625    ,
 0.00469970703125  ,
 0.028717041015625 ,
 0.0296478271484375,
 0.01220703125     ,
-0.009063720703125 ,
-0.0205535888671875,
-0.0170440673828125,
-0.0036163330078125,
 0.009735107421875 ,
 0.0146331787109375,
 0.0095977783203125,
-0.0006866455078125,
-0.008819580078125 ,
-0.0102081298828125,
-0.004852294921875 ,
 0.0027618408203125,
 0.0078125         ,
 0.0072174072265625,
 0.0021209716796875,
-0.0046539306640625,
-0.0092926025390625,
-0.0105438232421875,
-0.008331298828125 ,
-0.0050506591796875,
-0.0019989013671875
};
```

شکل 1-3. ضرایب coef

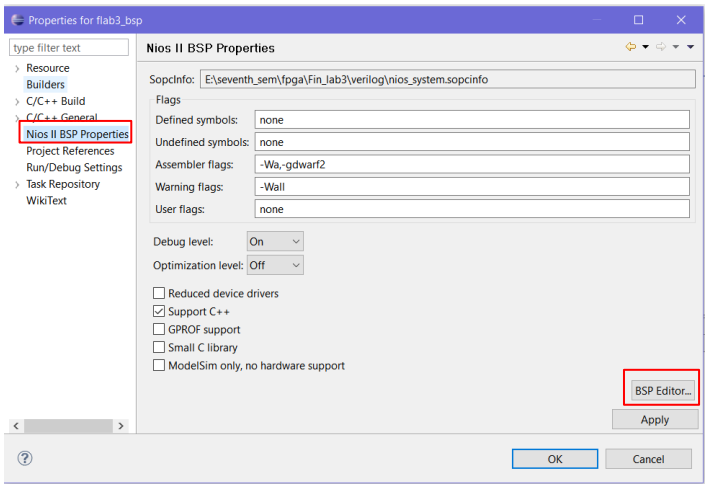
پس از اعمال فیلتر صدای پخش شده نویز صدای اولیه را نداشت ولی یک مقدار نویز دیگر به آن اضافه شده بود (حالت برفک به خود گرفته بود) که می توان به دلیل کم بودن دقت محاسبات در بخش نرم افزاری باشد.

همچنین زمان انجام محاسبات توسط توابع alt_timestamp_start و alt_timestamp محاسبه شد که این ها شمارنده کلاک می باشند و برای به دست آوردن زمان باید بر فرکانس تقسیم کرد که فرکانس را با استفاده از تابع alt_timestamp_freq به دست آوردیم. به این گونه که قبل از شروع کار فیلتر alt_timestamp_start را فراخوانی کردیم. قبل از فراخوانی فیلتر alt_timedtamp را دریک ذخیره کردیم و پس از فراخوانی فیلتر، مقدار alt_timestamp را در متغیری دیگر ذخیره کردیم. در نهایت اختلاف دو مقدار بدست آمده را بر این فرکانس تقسیم کرده تا زمان به دست آید و سپس آن را چاپ کردیم.

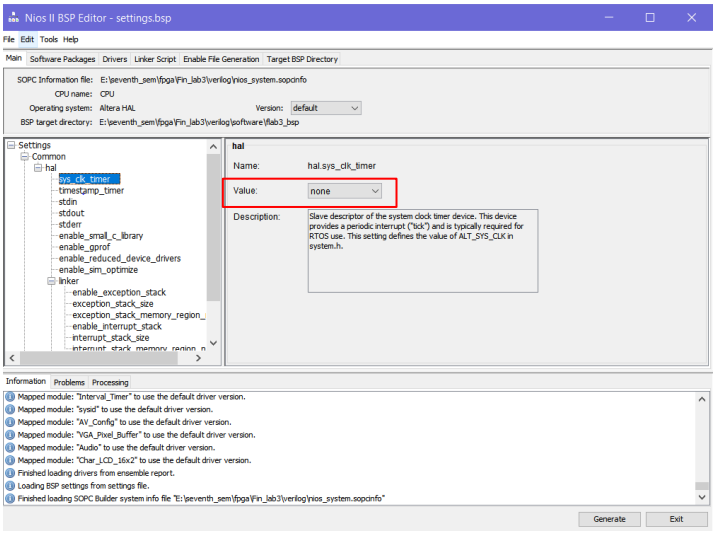
```
void run_command(struct alt_up_dev *up_dev) {
    int num_read;
    int num_written;
    if(command == 0){ //Record
        alt_u32 fr;
        fr = alt_timestamp_freq();
        // reset the buffer index for recording
        buf_index_record = 0;
        // clear audio FIFOs
        alt_up_audio_reset_audio_core (up_dev->audio_dev);
        // enable audio-in interrupts
        alt_up_audio_enable_read_interrupt (up_dev->audio_dev);
        alt_timestamp_start();
        float t_start = (float) alt_timestamp() / (float) fr;
        make_Denoise();
        float t_stop = (float) alt_timestamp() / (float) fr;
        printf("t = %f\n", t_stop - t_start);
```

شکل 1-4. محاسبه زمان اجرای فیلتر

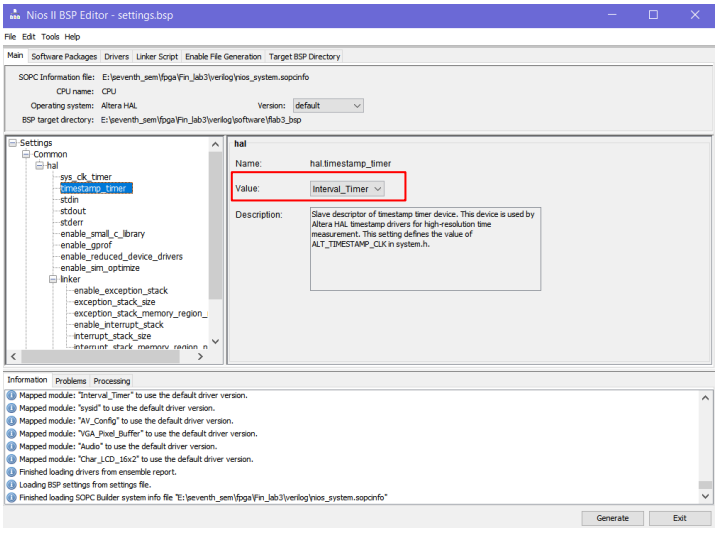
تنظیمات پروژه bsp را در eclipse مطابق مراحل دستورکار تنظیم کردیم. Timer بر روی none و timestamptimer بر روی interval_timer قرار گرفت تا بتوان از توبع زمانی که ذکر شد استفاده کرد.



شکل 1-5. تنظیمات timestamp

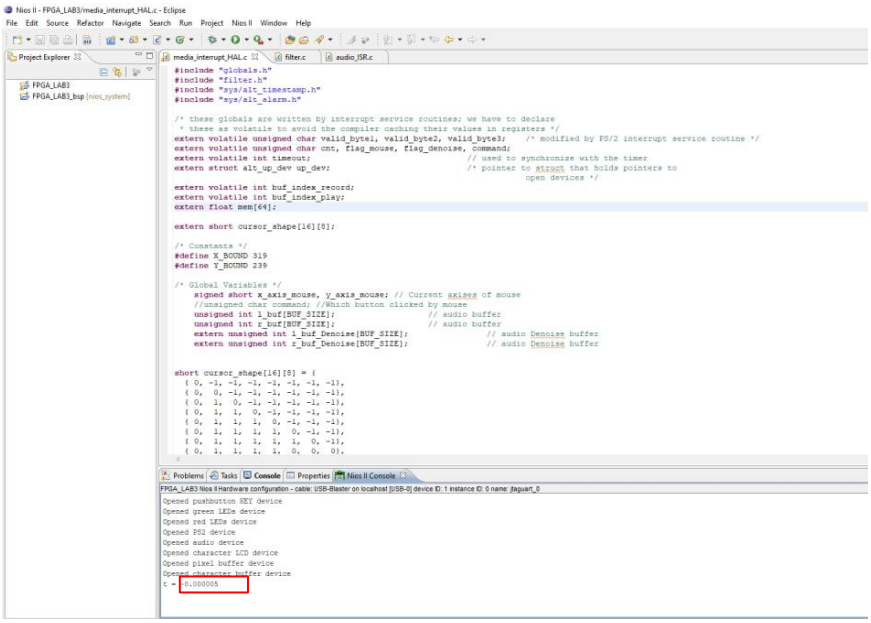


شکل 1-6. تنظیمات timestamp



شکل 1-7. تنظیمات timestamp

برای تست فیلتر، به دلیل تاخیر بسیار زیاد، نمونه های ضبط شده به 0.1 نمونه اولیه کاهش یافت در واقع یک ثانیه صدا ضبط شد و روی آن فیلتر اعمال شد. این کار 9 دقیقه و 50 ثانیه زمان برد که اگر 10 برابر میشد زمان ضبط شده این زمان به یک ساعت و نیم میرسید! همچنین با توجه به اینکه در این زمان شمارنده کلاک اورفلو میکند مقدار -0.000005 چاپ شد(حداکثر عدد قابل نمایش برابر =2^32 4,294,967,296 می باشد که این مقدار در صورتی که با فرکانس 50 مگاهرتز کار کنیم برابر با 86 ثانیه می شود.) که همان گونه که گفته شد این مقدار اورفلو شده می باشد و اشتباه است.



شکل 1-8. زمان محاسبه شده حالت نرم افزاری

گام 2: طراحی سخت افزاری فیلتر (ایجاد custom instruction جدید)

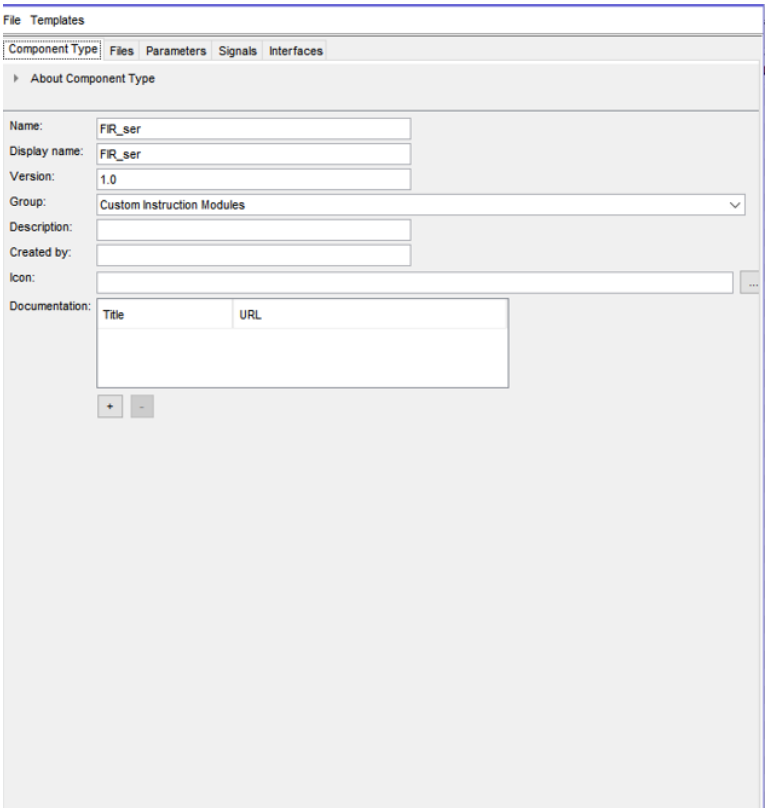
ابتدا تغییرات لازم را در کد وریلاگ تولید شده توسط متلب اعمال میکنیم:

```
module FIR_filter
               (
                clk,
                clk_enable,
                reset,
                input_,
                filter_out
                );

  input   clk;
  input   clk_enable;
  input   reset;
  input [31:0] input_;
  wire   signed [23:0] filter_in; //sfix24_En23
  output  signed [31:0] filter_out; //sfix32_En30

////////////////////////////////////////////////////////////////
//Module Architecture: FIR_filter
////////////////////////////////////////////////////////////////
  // Local Functions
  // Type Definitions
  // Constants
  assign filter_in = input_[31:8];
  localparam signed [15:0] coeff1 = 16'b1111111101111101; //sfix16_En16
  localparam signed [15:0] coeff2 = 16'b1111111010110101; //sfix16_En16
  localparam signed [15:0] coeff3 = 16'b1111110111011110; //sfix16_En16
  localparam signed [15:0] coeff4 = 16'b1111110101001101; //sfix16_En16
  localparam signed [15:0] coeff5 = 16'b1111110110011111; //sfix16_En16
  localparam signed [15:0] coeff6 = 16'b1111111011001111; //sfix16_En16
  localparam signed [15:0] coeff7 = 16'b0000000010001011; //sfix16_En16
  localparam signed [15:0] coeff8 = 16'b0000000111011001; //sfix16_En16
  localparam signed [15:0] coeff9 = 16'b0000001000000000; //sfix16_En16
  localparam signed [15:0] coeff10 = 16'b0000000010110101; //sfix16_En16
  localparam signed [15:0] coeff11 = 16'b1111111011000010; //sfix16_En16
```

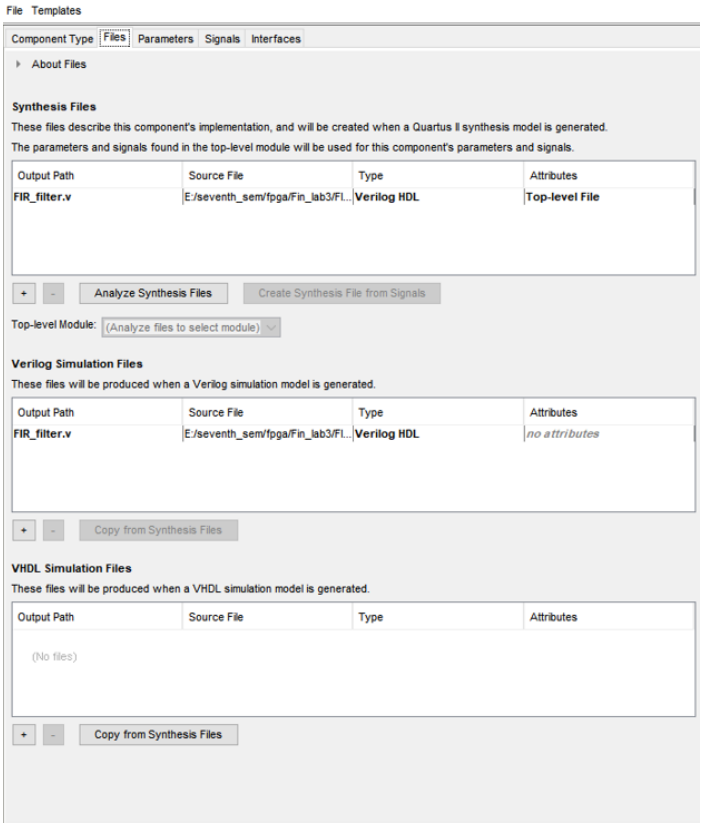
شکل 1-9. ادیت کد وریلاگ فیلتر

باید برای محاسبات فیلتر از 24 بیت پرارزش ورودی استفاده نماییم. همچنین تایپ ضرایب فیلتر را localparam signed قرار میدهیم.

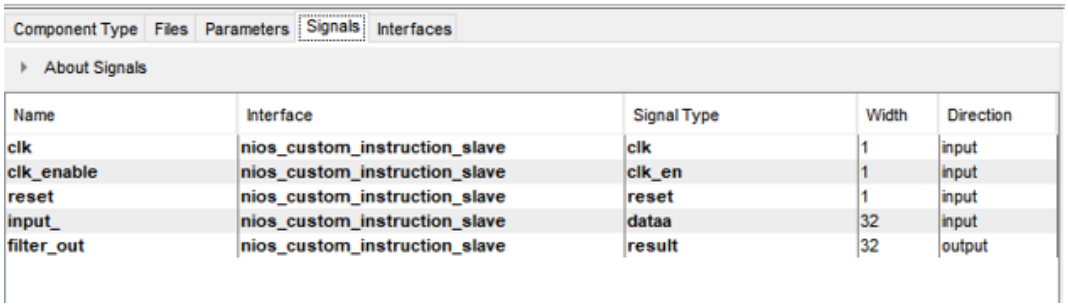
مراحل زیر را برای تولید یک custom instruction برای فیلتر در qsys طی میکنیم:



شکل 1-10. Component type



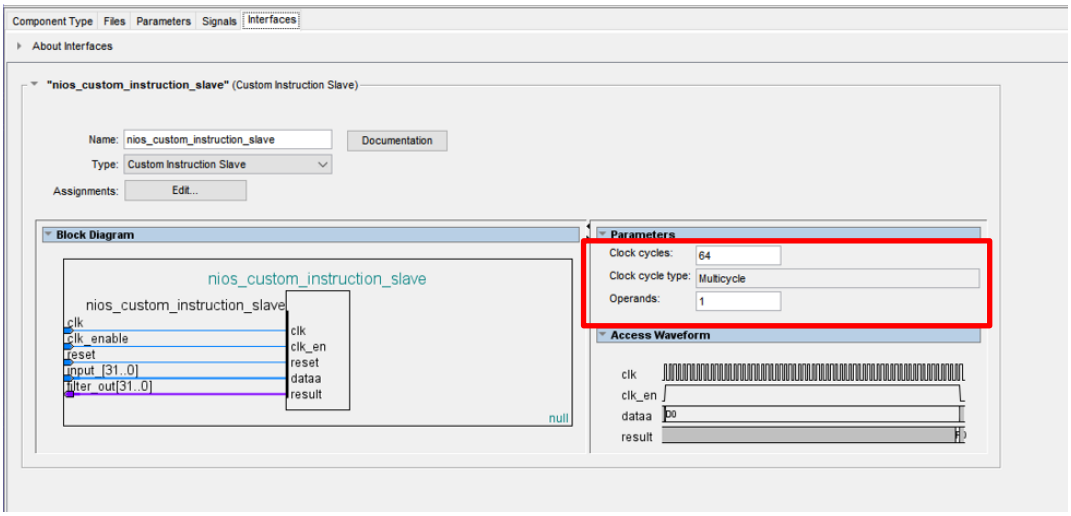
شکل 1-11. Files



Name	Interface	Signal Type	Width	Direction
clk	nios_custom_instruction_slave	clk	1	input
clk_enable	nios_custom_instruction_slave	clk_en	1	input
reset	nios_custom_instruction_slave	reset	1	input
input_	nios_custom_instruction_slave	dataa	32	input
filter_out	nios_custom_instruction_slave	result	32	output

شکل 1-12. Signals

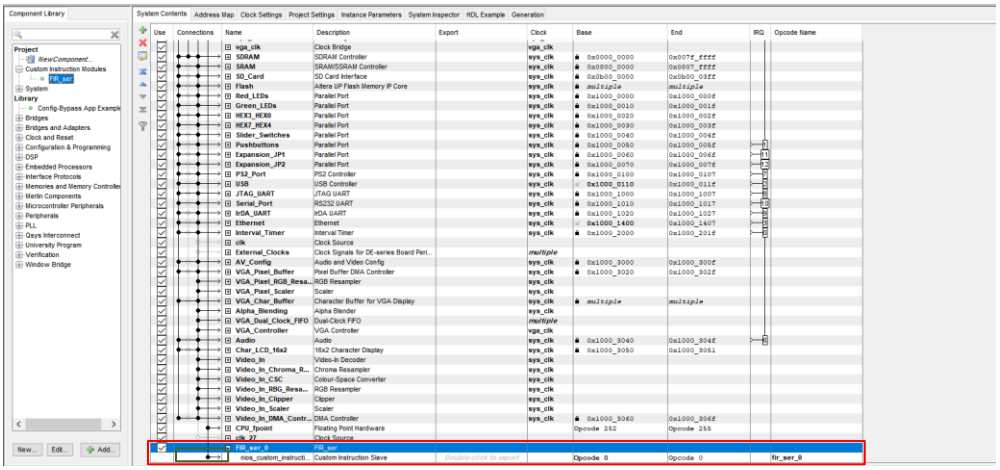
در بخش interface signals همه سیگنال ها را به nios_custom_instruction_slave تغییر داده و تایپ سیگنال ها را بر اساس نوعشان تنظیم کردیم.



شکل 1-13. interfaces

در بخش interface باید مقادیری که مشخص شده اند مطابق تصویر تغییر کنند. به دلیل اینکه درجه فیلتر 64 است، تعداد سیکل مورد نیاز را 64 و باتوجه به اینکه فقط یک ورودی داریم، operands را برابر 1 قرار میدهیم. Clk cycle type را هم multicycle قرار میدهیم؛ چون عملیات فیلتر به بیشتر از یک سیکل نیاز دارد.

پس از اعمال تغییرات custom instruction جدید به system contents اضافه میشود.

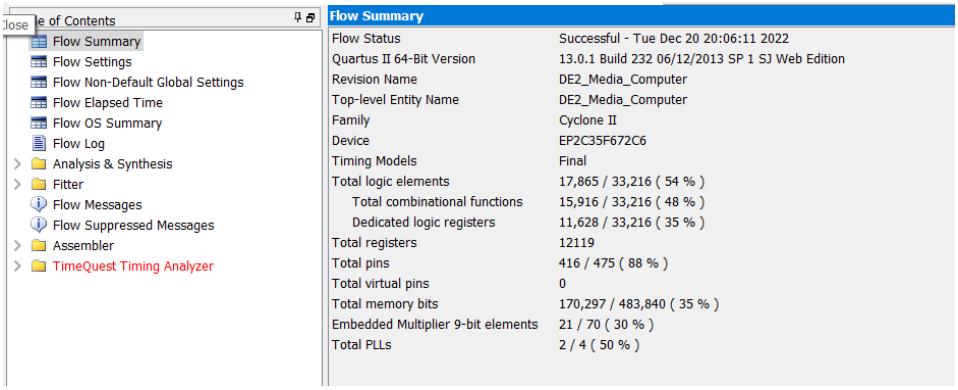


شکل 1-14. system contents

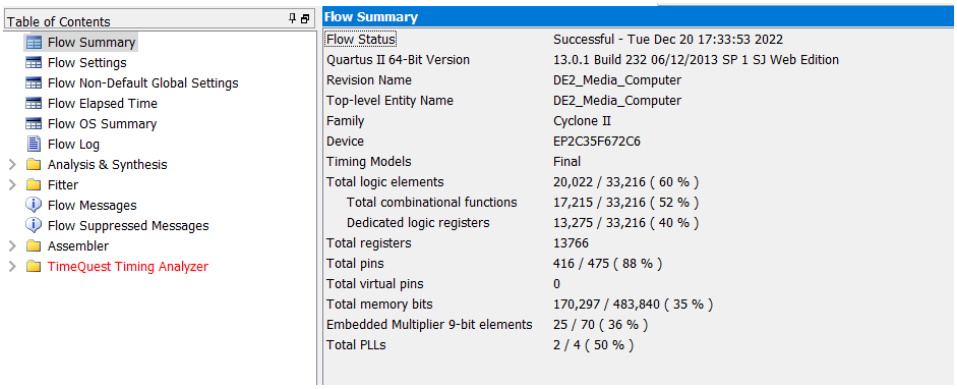
مطابق تصویر بالا، باید connection برای custom instruction جدید حتما ایجاد شود.

گام 3: استفاده از custom instruction تولید شده

پس از طی مراحل بالا کد را generate میکنیم و سپس بار دیگر سنتز میکنیم.



شکل 1-15. نتایج سنتز قبل از اضافه کردن custom instruction



شکل 1-16. نتایج سنتز بعد از اضافه کردن custom instruction

منابع	قبل از اضافه کردن custom instruction	بعد از اضافه کردن custom instruction
Total logic elements	17865	20022
Total registers	12119	13766
Embedded Multiplier 9-bit elements	21	25

جدول 1- مقایسه میزان منابع

پس از سنتز کردن، یک ماکرو برای custom instruction تولید شده در کد سخت افزار در قسمت system.h ایجاد میشود.

```
#define ALT_CI_FIR_SER_0(A) __builtin_custom_ini(ALT_CI_FIR_SER_0_N,(A))
#define ALT_CI_FIR_SER_0_N 0x0
```

شکل 1-17. New Macro

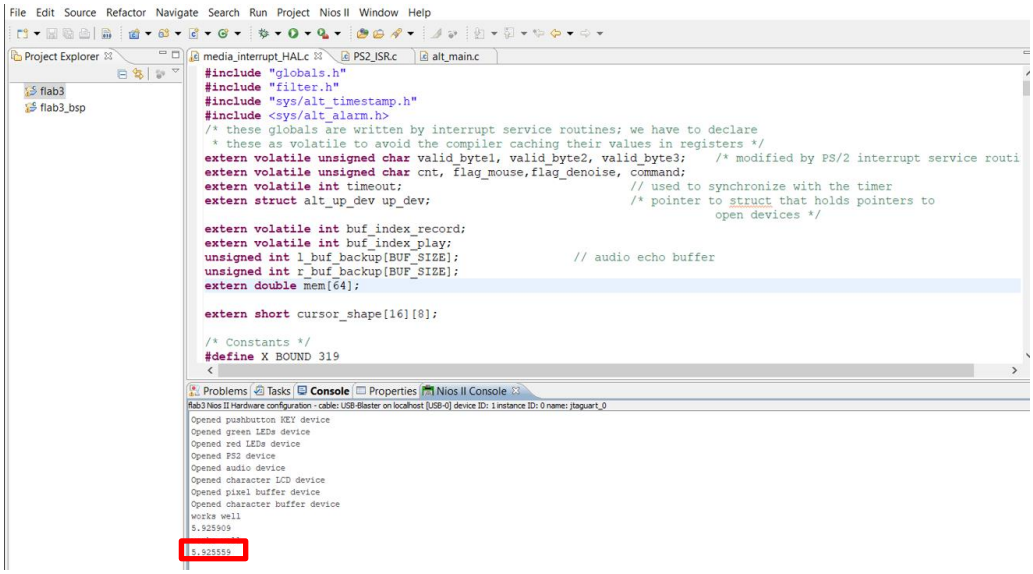
تغییرات زیر را در کد تابع make_denoise ایجاد میکنیم:

```
void make_Denoise(){
    int i;
    for (i = 0; i < BUF_SIZE; i++){
        l_buf_Denoise[i] = ALT_CI_FIR_SER_0(l_buf[i]);
    }
    for (i = 0; i < BUF_SIZE; i++){
        r_buf_Denoise[i] = ALT_CI_FIR_SER_0(r_buf[i]);
    }
}
```

شکل 1-18. دینویز کردن با ماکروی تولید شده

در طراحی سخت افزار پس از اجرای برنامه صدا ضبط شده و به سرعت دینویز شد. صدای دینویز شده با کیفیت خوبی بود و واضح شده بود و فیلتر بسیار خوب در بخش سخت افزاری کار کرد. به طوری که نویز موجود در صدای ضبط شده را به طور کامل برطرف کرد و دیگر نویزی قابل شنیدن نبود.

مطابق حالت نرم افزاری با استفاده از timestamp زمان اجرای فیلتر را محاسبه میکنیم.



شکل 1-19. زمان اجرای فیلتر

زمان اندازه گیری شده مطابق تصویر، 5.925559 ثانیه است که واضحا از فیلتر نرم افزاری بسیار سریع تر است.

از مقایسه زمان سخت افزار و نرم افزار در میابیم که سرعت طراحی سخت افزاری خیلی بیشتر بوده و در حد چند ثانیه است. در صورتی که زمان نرم افزار به ساعت میرسد. پس طراحی سخت افزاری هر چند ممکن است سخت تر باشد ولی از نظر زمان و توان بسیار بهتر عمل میکند.

سوال: نیازمندی های تعریف یک دستور Multicycle

برای تعریف دستور multicycle باید دید آیا دستوری که میخواهیم اضافه کنیم در یک تعداد سیکل مشخص عملیات آن به پایان میرسد یا تعداد سیکل انجام عملیات آن متغیر است. اگر تعداد سیکل مشخص باشد در generation system تعداد سیکل مورد نیاز دستور را تعیین میکنیم. اگر تعداد سیکل های اجرای دستور متغیر باشد، با استفاده از تعریف دو پورت جدید start و done شروع و پایان انجام عملیات دستور را با استفاده از مکانیزم handshaking انجام میدهیم .در تعریف custom instruction ها، پورت هایی میتواند تعیین شود که برای multicycle custom instruction ها پورت های clk, clk_en, rst اجباری هستند. برای دستور با تعداد سیکل های متغیر پورت های start, done هم نیاز است. در شکل 1-8. Signals پورت های تعریف شده برای custom instruction موردنیاز ما قابل مشاهده است.