به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

مدارهای مبتنی بر FPGA

آزمایش شماره 3

علیرضا جابری راد

نیما سلیمی

پاییز 1400

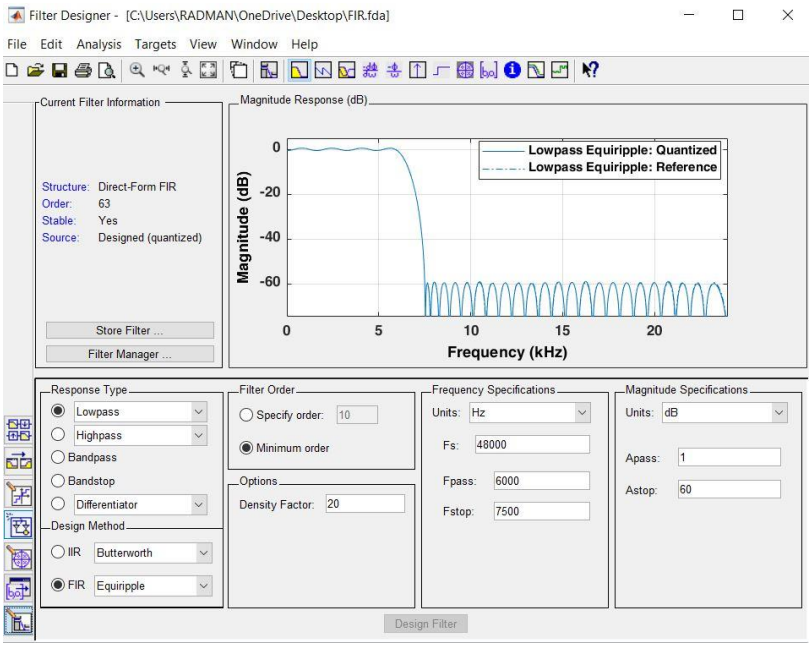
فهرست

چکیده

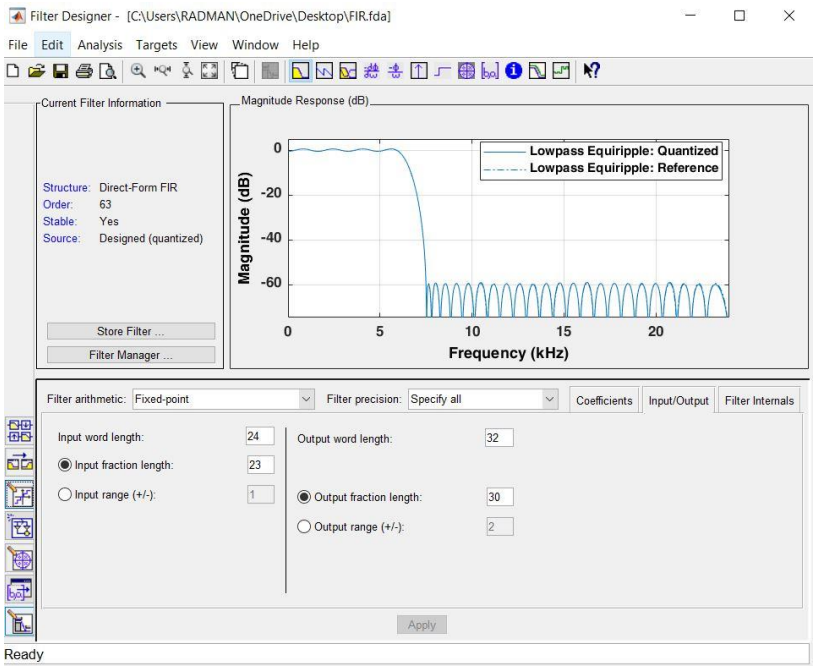
در این آزمایش قصد داریم فیلتری در محیط متلب برای فیلتر کردن یک ویس 10 ثانیه ای طراحی کنیم که بخش اول این آزمایش در محیط متلب انجام می شود سپس با استخراج کد های RTL مربوطه و تست بنچ ها در دو حالت کاملا موازی و کاملا سری به سراغ نرم افزارهای مربوطه و در بخش آخر برد DE2 می رویم تا نویز این صدا را برداشته و صدای بدون نویز را بشنویم.

بخش نرم افزار MATLAB

تمام مراحل طراحی سیستم را با استفاده از دستور flterDesigner طراحی کردیم که در زیر تصاویر مربوطه آورده شده است:



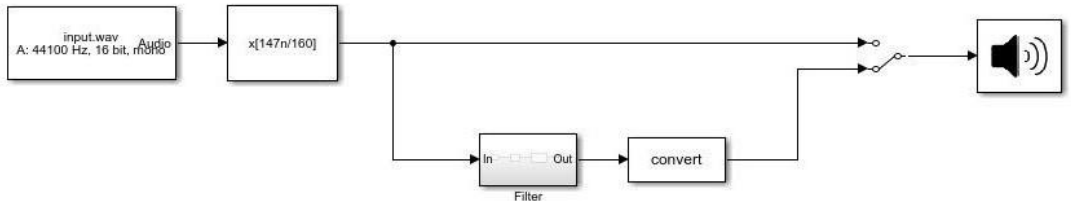
شکل1)تغییر پارامترهای فیلتر مطابق خواسته



شکل2) تغییر طول ورودی ها و خروجی ها

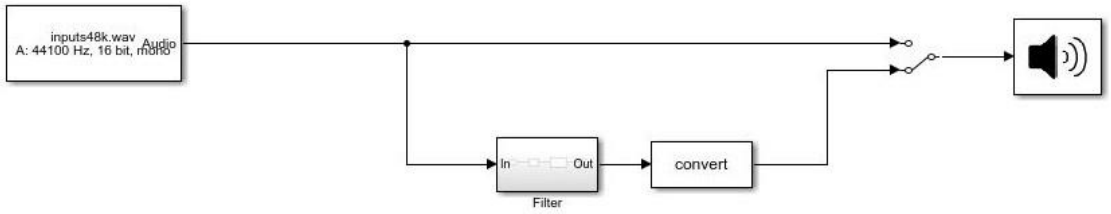
نکته: در اینجا با اشتباه وارد کردن اندازه فیلتر (51) باعث خراب شدن صدا در خروجی آخر شدیم که با تصحیح این طول به (64) خروجی مطلوب حاصل شد.

بخش سیمولینک)



شکل3) شبیه سازی سیمولینک از شکل فیلتر مورد استفاده مطابق شکل 3 در گزارشکار و یکی از خواسته ی آزمایش

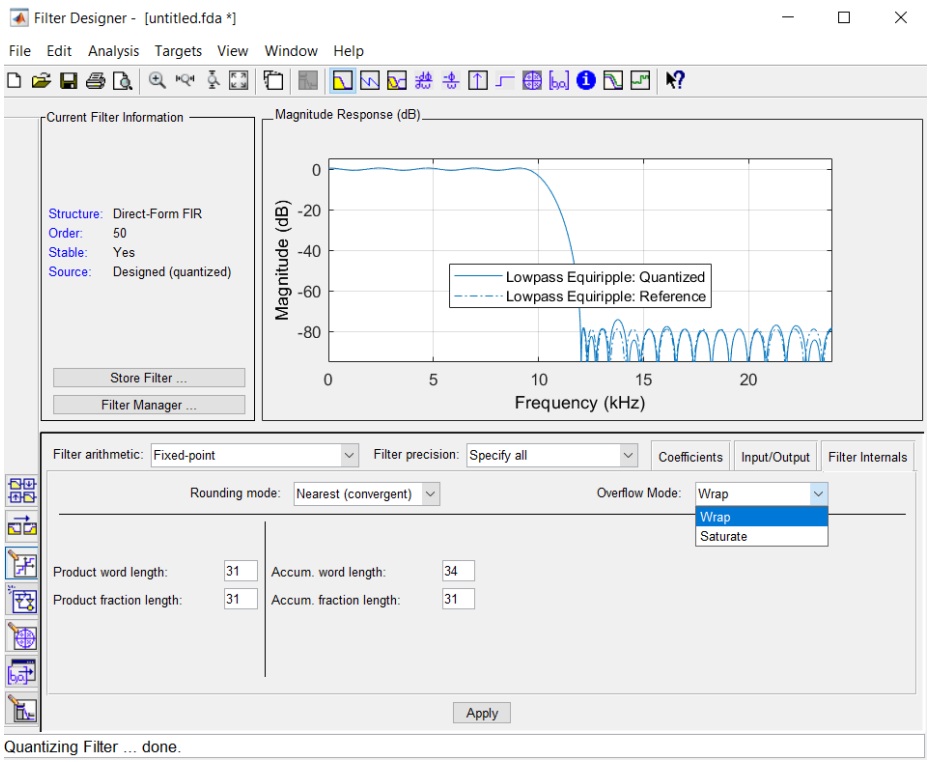
در بخش قبل و محیط Simulink مجبور بودیم بلوک x[147/160] را برای امر resampling بعد از بلوک ورودی قرار دهیم ولی در این بخش قصد داریم در محیط متلب با اجرای کد resample و با دادن اعداد 147 و 160 بدون استفاده از بلوک resampling این کار را انجام دهیم با این تفاوت که حال ورودی به جای فایل input.wav همان آرایه inputs48k خواهد بود.



شکل4) شبیه سازی سیمولینک از فیلتر با ورودی resample شده

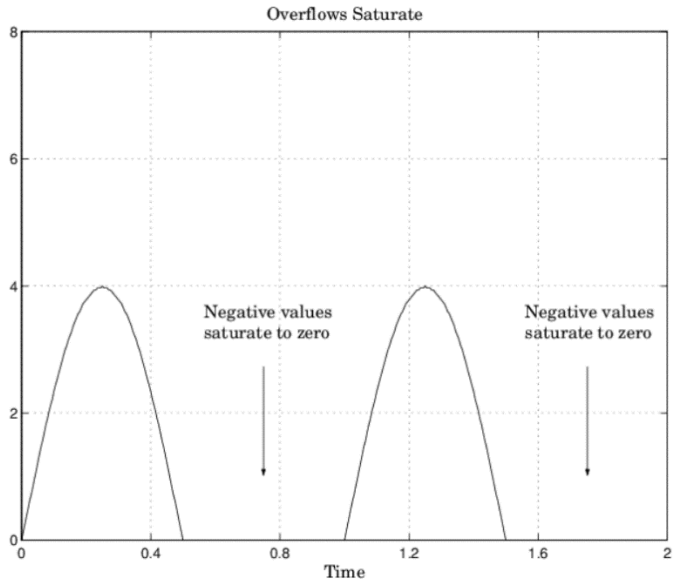
تفاوت بین حالت Wrap و Saturate :

پردازنده ها سیاست های خاصی برای برخورد با سرریز را دارند و Wrapping و Saturation هر دو برای مقابله با Overflow ساخته شده اند. و معمولا پردازنده ها از هر دو حالت پشتیبانی می کنند.در محیط filterDesigner بعد از اینکه filter arithmetic را روی Fixed_point و در قسمت filter precision ، specify all را انتخاب کردیم در پنجره filter Internals می توان حالت سرریز را انتخاب کرد:

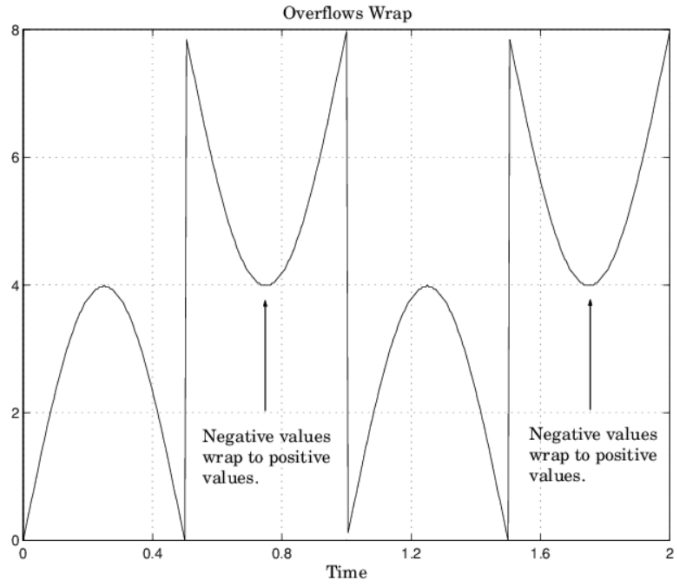


شکل5) پنجره Filter Internals برای انتخاب wrap یا saturate

مثلا در حالت Saturation اعداد منفی را به صفر (کوچک ترین عدد قابل نمایش) تناظر می کند و اعداد مثبت را به همان شکل نشان می دهد. در حالت Wrapping برای مقابله Overflow اعداد منفی به اعداد مثبت مناسب متناظر می شوند به نحوی که با اعداد مثبت اشتباه گرفته نشوند. در زیر می توان تفاوت این دو حالت را مشاهده کرد. این شکل ها از سایت Mathworks.com برای مثال تابع سینوسی متناوب [4 -4] کشیده شده اند:



شکل6) مثالی از حالت Saturation



شکل7) مثالی از حالت Wrapping

دلیل کوانتیزاسیون ضرایب:

فرایند دیجیتالی کردن یک سیگنال آنالوگ شامل گرد کردن مقادیری است که تقریبا برابر با مقدار آنالوگ هستند. با استفاده از نمونه‌برداری چند نقطه روی سیگنال آنالوگ اصلی انتخاب می‌شود و سپس این نقاط پس از گرد کردن مقادیر به نزدیک‌ترین مقدار پایدار، به یکدیگر متصل می‌شوند. به این فرایند «کوانتیزاسیون» (Quantization)می‌گویند .

کوانتیزاسیون نشان دهنده این است که مقادیر دامنه نمونه برداری شده عضو یک مجموعه محدود از مقادیر هستند. این امر به معنی تبدیل یک نمونه پیوسته در زمان به یک نمونه گسسته در زمان تلقی می‌شود. فرایند نمونه برداری از یک سیگنال و نیز کوانتیزاسیون آن در واقع منجر به از دست رفتن اطلاعات می‌شود. کیفیت خروجی یک فرایند کوانتیزاسیون به تعداد سطوح کوانتیزاسیون مورد استفاده بستگی دارد. در مورد سیگنال‌های آنالوگ صوتی، هنگام کوانتیزاسیون و تبدیل آن‌ها به یک مقدار دیجیتال، معمولا یک نوع خطای کوانتیزاسیون اتفاق می‌افتد. این چنین خطاهایی، یک نویز با باند بسیار بزرگ به وجود می‌آورند که نویز کوانتیزاسیون نام دارد.

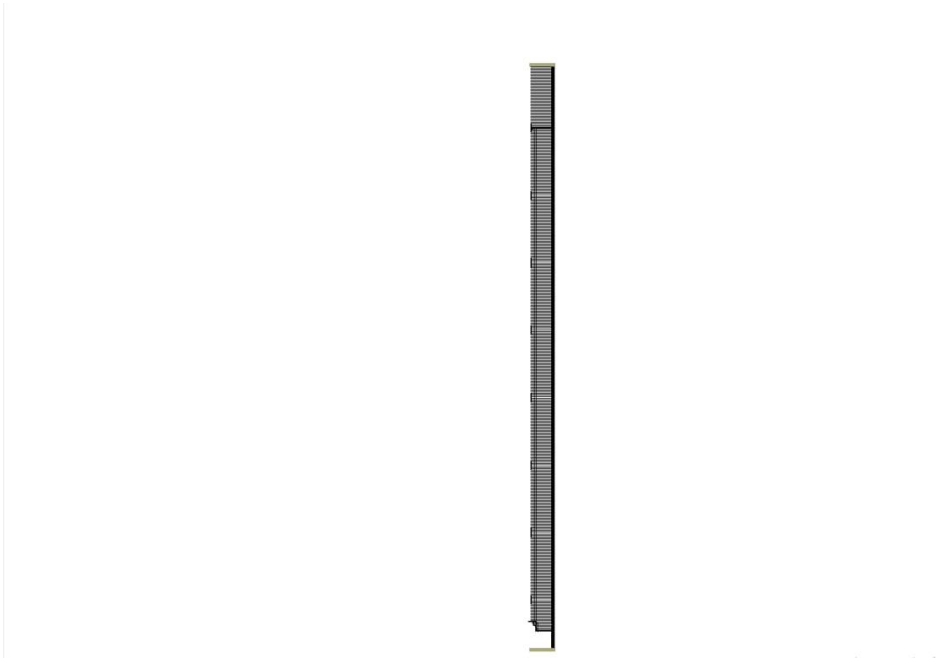
در نهایت چون که تعداد بیت های مورد استفاده عددی محدود بوده و می دانیم که ضرایب نباید به صورت floating point ذخیره شوند و با کوانتیزه کردن ضرایب خطایی حاصل می شود که خطای کوانتیزاسیون نام دارد ولی مزیت این کار این است که خطا کمترین خطای ممکن است. با اصطلاح ضرایب فیکس می شوند و تضعیف در این حالت کمتر می شود.

توصیف سطح بالای فیلتر در نرم افزار کوارتوس:

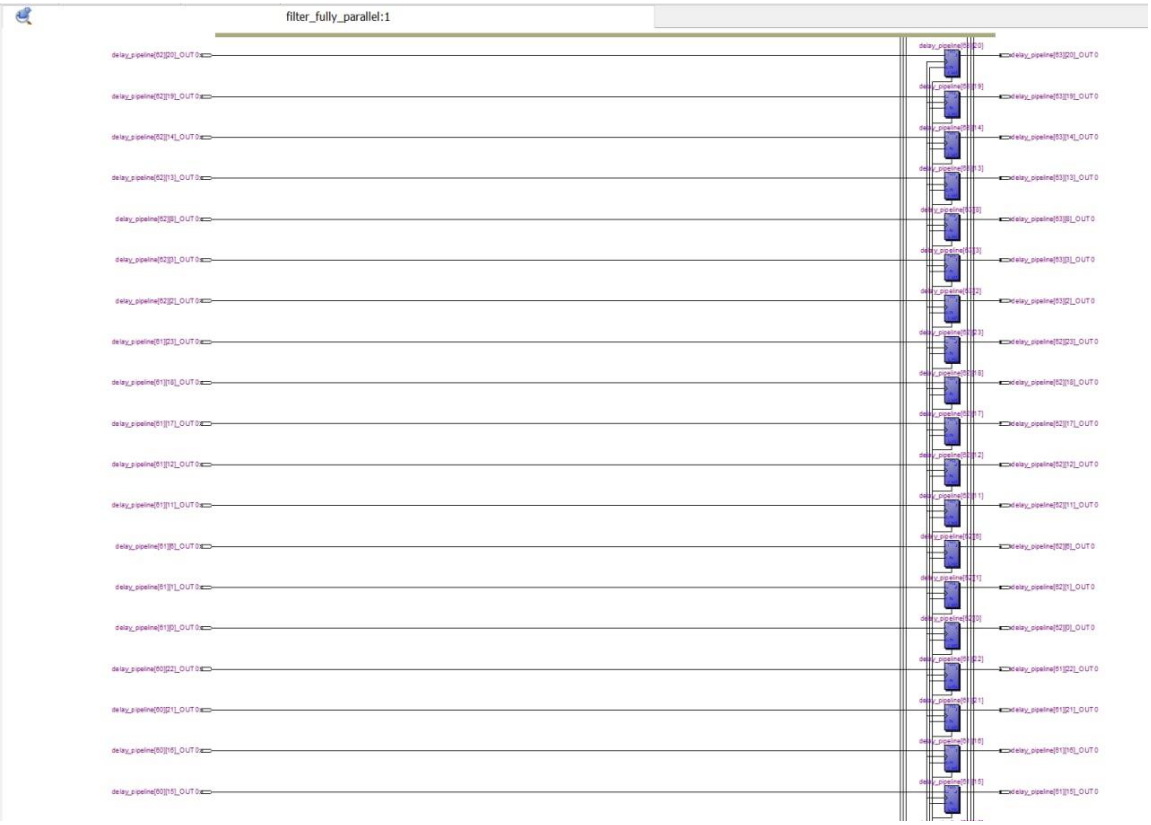
در قسمت generate hdl… در نرم افزار متلب می توان دو کد کاملا موازی و کاملا سریال فیلتر را انتخاب کرد و سپس کد وریلاگ هر کدام و کد تست بنچ را generate کرد البته در قسمتی که آرایه inputs48k را به عنوان ورودی دادیم نیز مد کاملا سریال فیلتر و تست بنچ مربوط به آن را generate کردیم ولی به علت اهمیت بالا در زیر فقط دو کد وریلاگ کاملا موازی و کاملا سریال قسمت قبلی (با ورودی input.wav و بلوک نمونه برداری آورده شده است:

تذکر: توضیحات کدها و تست بنچ مربوطه در قسمت بعد آورده شده اند.

Fully Parallel :



شکل 8) نمای دور RTL مربوط به فیلتر Fully Parallel



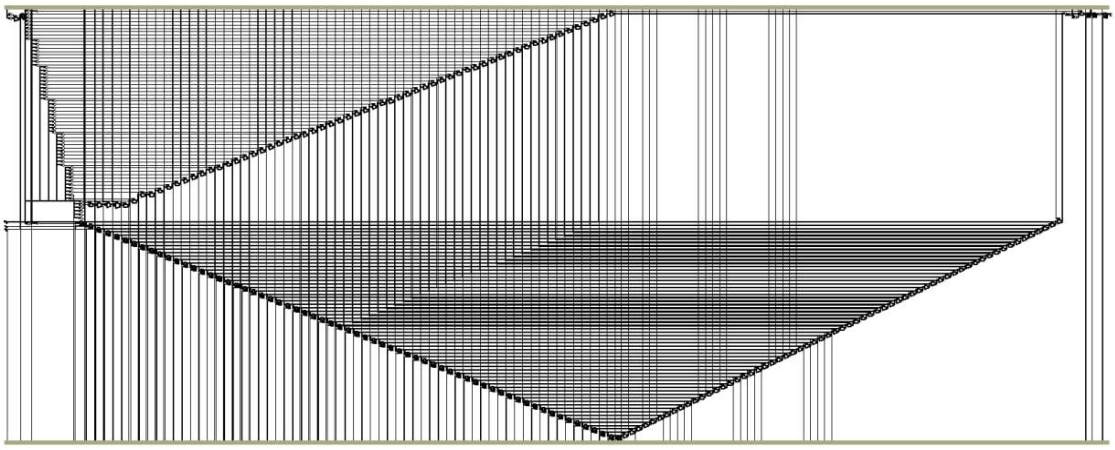
شکل 9) زوم شده شکل8

نتیجه سنتز کد تماما موازی در کوارتوس:

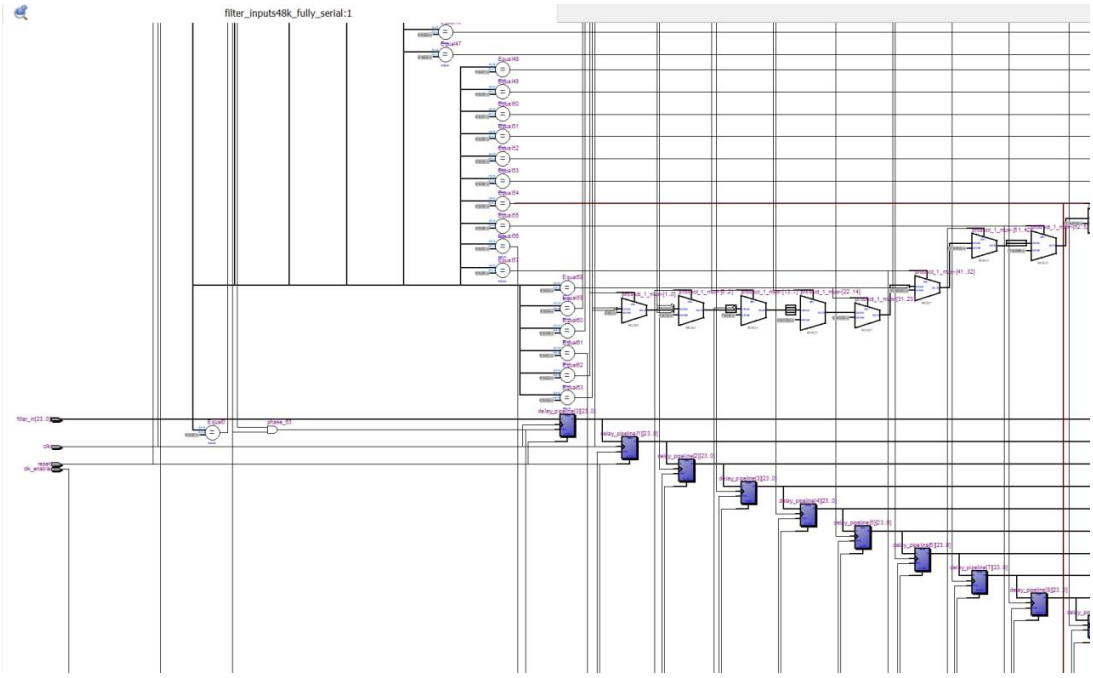
Flow Summary	
Flow Status	Successful - Wed Dec 22 00:38:10 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	filter_fully_parallel
Top-level Entity Name	filter_fully_parallel
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	16,993 / 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 %)

شکل10) نتیجه سنتز کد Fully Parallel در کوارتوس

Fully Serial :



شکل11) شماتیک کلی **RTL** مربوط به کد Fully Serial



شکل 12) زوم شده شکل 11

نتیجه سنتز کد Fully Serial در نرم افزار کوارتوس:

Flow Summary	
Flow Status	Successful - Wed Dec 22 00:45:15 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	filter_inputs48k_fully_serial
Top-level Entity Name	filter_inputs48k_fully_serial
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
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 %)

شکل 13) نتیجه سنتز fully serial در کوارتوس

تحلیل کد های وریلاگ

حالت تماما موازی:

ورودی و خروجی های فیلتر موازی به شکل زیر است:

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

تعریف پارامتر های ماژول:

```
parameter signed [15:0] coeff1 = 16'b1111111101111101; //sfix16_En16
parameter signed [15:0] coeff2 = 16'b1111111010110101; //sfix16_En16
```

در این حالت 64 ضرب کننده و جمع کننده داریم.

سیگنال های کنترلی:

```
// Signals
reg  signed [23:0] delay_pipeline [0:63] ; // sfix24_En23
wire signed [30:0] product64; // sfix31_En31
wire signed [39:0] mul_temp; // sfix40_En39
```

رجیستر اول برای تاخیر استفاده می شود.

وایر ها ورودی ضرب کننده ها هستند.

برای تاخیر ها داریم:

```
// Block Statements
always @( posedge clk or posedge reset)
  begin: Delay_Pipeline_process
    if (reset == 1'b1) begin
      delay_pipeline[0] <= 0;
      delay_pipeline[1] <= 0;
      delay_pipeline[2] <= 0;
```

ضرب کننده ها و جمع کننده ها در زیر نشان داده شدند:

```
assign mul_temp = delay_pipeline[63] * coeff64;
assign product64 = (mul_temp[38:0] + {mul_temp[8], {7{~mul_temp[8]}}})>>>8;
```

x64

عملیات فیلتر کلی و جمع نمونه ها به صورت پلکانی:

```
assign add_signext = product1_cast;
assign add_signext_1 = $signed({{3{product2[30]}}, product2});
assign add_temp = add_signext + add_signext_1;
assign sum1 = add_temp[33:0];

assign add_signext_2 = sum1;
assign add_signext_3 = $signed({{3{product3[30]}}, product3});
assign add_temp_1 = add_signext_2 + add_signext_3;
assign sum2 = add_temp_1[33:0];
```

آخرین رجیستر و خروجی:

```
  assign output_typeconvert = (sum63[32:0] + sum63[1])>>>1;

  always @ (posedge clk or posedge reset)
    begin: Output_Register_process
      if (reset == 1'b1) begin
        output_register <= 0;
      end
      else begin
        if (clk_enable == 1'b1) begin
          output_register <= output_typeconvert;
        end
      end
    end // Output_Register_process

  // Assignment Statements
  assign filter_out = output_register;
endmodule  // filter fully parallel
```

ابتدا ورودی رجیستر مشخص شده و سپس به رجیستر می رود که با کلاک و ریست آسنکرون کار می کند سپس خروجی در filter_out ریخته می شود.

کد تماما سری:

ورودی ها و خروجی ها به شکل زیر می باشند:

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

رجیستر ها و وایر های تعریف شده:

```
reg  signed [23:0] delay_pipeline [0:63] ; // sfix24_En23
wire signed [23:0] inputmux_1; // sfix24_En23
reg  signed [33:0] acc_final; // sfix34_En31
reg  signed [33:0] acc_out_1; // sfix34_En31
wire signed [30:0] product_1; // sfix31_En31
wire signed [15:0] product_1_mux; // sfix16_En16
```

رجیستر اول برای ایجاد تاخیر ، وایر اول همانطور که از اسمش پیداست ورودی ضرب کننده است و وایر آخر دیگر ورودی آن است و رجیستر دوم و سوم خروجی آن هستند.

در اینجا پارامترها تعریف شده اند:

```
parameter signed [15:0] coeff1 = 16'b1111111101111101; //sfix16_En16
parameter signed [15:0] coeff2 = 16'b1111111010110101; //sfix16_En16
parameter signed [15:0] coeff3 = 16'b1111110111011110; //sfix16_En16
parameter signed [15:0] coeff4 = 16'b1111110101001101; //sfix16_En16
parameter signed [15:0] coeff5 = 16'b1111110110011111; //sfix16_En16
```

کنترل مدار به عهده این counter است:

```
// Block Statements
always @ (posedge clk or posedge reset)
  begin: Counter_process
    if (reset == 1'b1) begin
      cur_count <= 6'b111111;
    end
    else begin
      if (clk_enable == 1'b1) begin
        if (cur_count >= 6'b111111) begin
          cur_count <= 6'b000000;
        end
        else begin
          cur_count <= cur_count + 6'b000001;
        end
      end
    end
  end // Counter_process
```

این سیگنال های کنترلی مربوط به شمارنده هستند:

```
assign  phase_63 = (cur_count == 6'b111111 && clk_enable == 1'b1) ? 1'b1 : 1'b0;

assign  phase_0 = (cur_count == 6'b000000 && clk_enable == 1'b1) ? 1'b1 : 1'b0;
```

رجیستر های تاخیر:

```
  always @( posedge clk or posedge reset)
    begin: Delay_Pipeline_process
      if (reset == 1'b1) begin
        delay_pipeline[0] <= 0;
        delay_pipeline[1] <= 0;
        delay_pipeline[2] <= 0;
        delay_pipeline[3] <= 0;
        delay_pipeline[4] <= 0;
        delay_pipeline[5] <= 0;
        delay_pipeline[6] <= 0;
        delay_pipeline[7] <= 0;
        delay_pipeline[8] <= 0;
```

ورودی فیلتر:

```
   assign inputmux_1 = (cur_count == 6'b000000) ? delay_pipeline[0] :
                       (cur_count == 6'b000001) ? delay_pipeline[1] :
                       (cur_count == 6'b000010) ? delay_pipeline[2] :
                       (cur_count == 6'b000011) ? delay_pipeline[3] :
                       (cur_count == 6'b000100) ? delay_pipeline[4] :
                       (cur_count == 6'b000101) ? delay_pipeline[5] :
```

ضریب فیلتر:

```
          assign product_1_mux = (cur_count == 6'b000000) ? coeff1 :
                                 (cur_count == 6'b000001) ? coeff2 :
                                 (cur_count == 6'b000010) ? coeff3 :
                                 (cur_count == 6'b000011) ? coeff4 :
                                 (cur_count == 6'b000100) ? coeff5 :
                                 (cur_count == 6'b000101) ? coeff6 :
```

دور ریختن بیت ها :

```
    assign mul_temp = inputmux_1 * product_1_mux;
    assign product_1 = (mul_temp[38:0] + {mul_temp[8], {7{~mul_temp[8]}}})>>>8;

    assign prod_typeconvert_1 = $signed({{3{product_1[30]}}, product_1});

    assign add_signext = prod_typeconvert_1;
    assign add_signext_1 = acc_out_1;
    assign add_temp = add_signext + add_signext_1;
    assign acc_sum_1 = add_temp[33:0];

    assign acc_in_1 = (phase_0 == 1'b1) ? prod_typeconvert_1 :
                      acc_sum_1;
```

جمع کننده همراه با سه رجیسترش:

```
always @ (posedge clk or posedge reset)
  begin: Acc_reg_1_process
    if (reset == 1'b1) begin
      acc_out_1 <= 0;
    end
    else begin
      if (clk_enable == 1'b1) begin
        acc_out_1 <= acc_in_1;
      end
    end
  end // Acc_reg_1_process

always @ (posedge clk or posedge reset)
  begin: Finalsum_reg_process
    if (reset == 1'b1) begin
      acc_final <= 0;
    end
    else begin
      if (phase_0 == 1'b1) begin
        acc_final <= acc_out_1;
      end
    end
  end // Finalsum_reg_process

assign output_typeconvert = (acc_final[32:0] + acc_final[1])>>>1;
```

و در نهایت خروجی به دست می آید:

```
  // Assignment Statements
  assign filter_out = output_register;
endmodule  // filter_fully_serial
```

عملکرد کد testbench

تست بنچ تولید شده توسط متلب به این صورت عمل میکند که پس از اجرا شدن، مقادیر ورودی از پیش تولید شده را به ورودی فیلتر میدهد و از سوی دیگر مقادیر خروجی فیلتر را با مقادیر مورد انتظار مقایسه میکند و در صورت عدم تطابق ارور میدهد. شبیه سازی تست بنچ به صورتی است که بلافاصله بعد از مشاهده ی عدم تطابق ارور میدهد. به ازای هر ارور خروجی دستور زیر در کنسول نمایش داده میشود

```
$display("ERROR in filter_out at time %t : Expected '%h' Actual '%h'",
     $time, filter_out_expected[filter_out_addr], filter_out);
```

اگر تعداد ارور ها از 3520 ارور بیشتر شد پیغام خروجی دستور زیر در کنسول نمایش داده میشود

```
if (filter_out_errCnt >= MAX_ERROR_COUNT)
  $display("Warning: Number of errors for filter_out have exceeded the maximum error limit");
```

در پایان اگر تست بنچ به پایان برسد snkDone یک میشود و با استفاده از always block زیر وجود یا عدم وجود ارور در عملکرد فیلتر اطلاع داده میشود:

```
always @(posedge clk or posedge reset) // completed_msg
begin
  if (reset) begin
     // Nothing to reset.
  end
  else begin
    if (snkDone == 1) begin
      if (testFailure == 0)
            $display("**************TEST COMPLETED (PASSED)**************");
      else
            $display("**************TEST COMPLETED (FAILED)**************");
    end
  end
end // completed_msg;
```

بخش 2: اضافه کردن دستور اختصاصی به پردازنده Nios II

گام اول

همانطور که در تصویر زیر میبینید، کد echo کردن، کامنت شده و محاسبات نرم افزاری مربوط به فیلتر، جایگزین آن شده است:

```
void make_echo(){
    int i,j;
    double accumulator_l, accumulator_r;
    /*for(i=5000;i<BUF_SIZE;i++){
        l_buf_echo[i]=l_buf[i-5000]+l_buf[i-1000];
        r_buf_echo[i]=r_buf[i-5000]+r_buf[i-1000];
    }*/
    /*for(i=0;i<BUF_SIZE/5;i++)//custom instruction
        l_buf_echo[i]=ALT_CI_FIR_0(l_buf[i]);
    for(i=0;i<BUF_SIZE/5;i++)
        r_buf_echo[i]=ALT_CI_FIR_0(r_buf[i]);*/
    for(i=0;i<BUF_SIZE;i++){//software FIR computation
        printf("%d\n",i);
        accumulator_l=0;
        accumulator_r=0;
        for(j=i;j>=0 && i-j<64;j--){
            accumulator_l=accumulator_l+(double)(l_buf[j]>>8)*coeffs[i-j];
            accumulator_r=accumulator_r+(double)(r_buf[j]>>8)*coeffs[i-j];
        }
        l_buf_echo[i]=((unsigned int)(accumulator_l));
        r_buf_echo[i]=((unsigned int)(accumulator_r));
    }
    return;
}
```

با توجه به زمان اندازه گیری شده، زمان انجام محاسبات به ازای هر 10000 خروجی فیلتر حدودا برابر 1 دقیقه بوده و جهت انجام فیلتراسیون روی یک صوت 10 ثانیه ای، حدود 50 دقیقه زمان نیاز است!

ثبت زمان در این آزمایش با استفاده از timestamp که مبتنی بر کلاک سخت افزاری است انجام شده است که تصویر کد آن را در زیر مشاهده میکنید:

```
if(make_echo_flag){
    alt_timestamp_start();
    printf("Filtering Calculation Started...\n");
    make_echo();
    make_echo_flag=0;
    calculation_time=(float)alt_timestamp()/(float)timestamp_freq;
    printf("Filtering Calculation Finished in %.3f seconds\n", calculation_time);
}
```

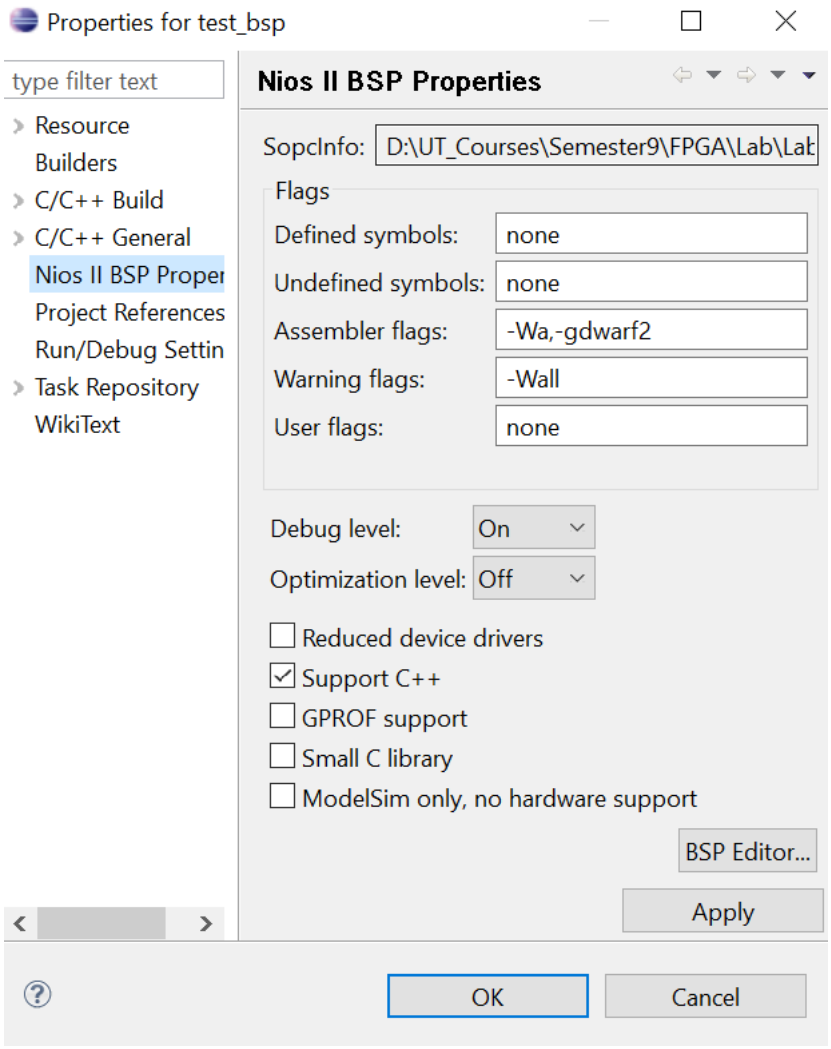
در این کد، به محض یک شدن flag مربوط به انجام محاسبات اکو(که در این آزمایش به فیلتر پایین گذر تبدیل شده است) محاسبات آغاز میشود و شمارنده ی timestamp با دستور alt_timestamp_start() ریست میشود و شروع به شمارش میکند. پس از انجام محاسبات توسط تابع make_echo()، flag ای که توسط ISR یک شده بود را صفر میکنیم تا محاسبات در حلقه های بعد، بدون آنکه وقفه ای بیاید مجددا انجام نشود. در خط بعد نیز با صدور دستور alt_timestamp آخرین عدد ثبت شده توسط شمارنده timestamp برگردانده میشود و با تقسیم آن عدد بر فرکانس کلاک timestamp که در متغیر timestamp_freq ذخیره کردیم، زمان انجام عملیات بر حسب ثانیه بدست میاید و در خط پایانی آنرا

گزارش میکنیم. در استفاده از timestamp برای ثبت زمان، چون عملیات به صورت سخت افزاری انجام میشود، لازم است تنظیمات پیش فرض پروژه به منظور راه اندازی شمارنده ی timestamp تغییر کند.

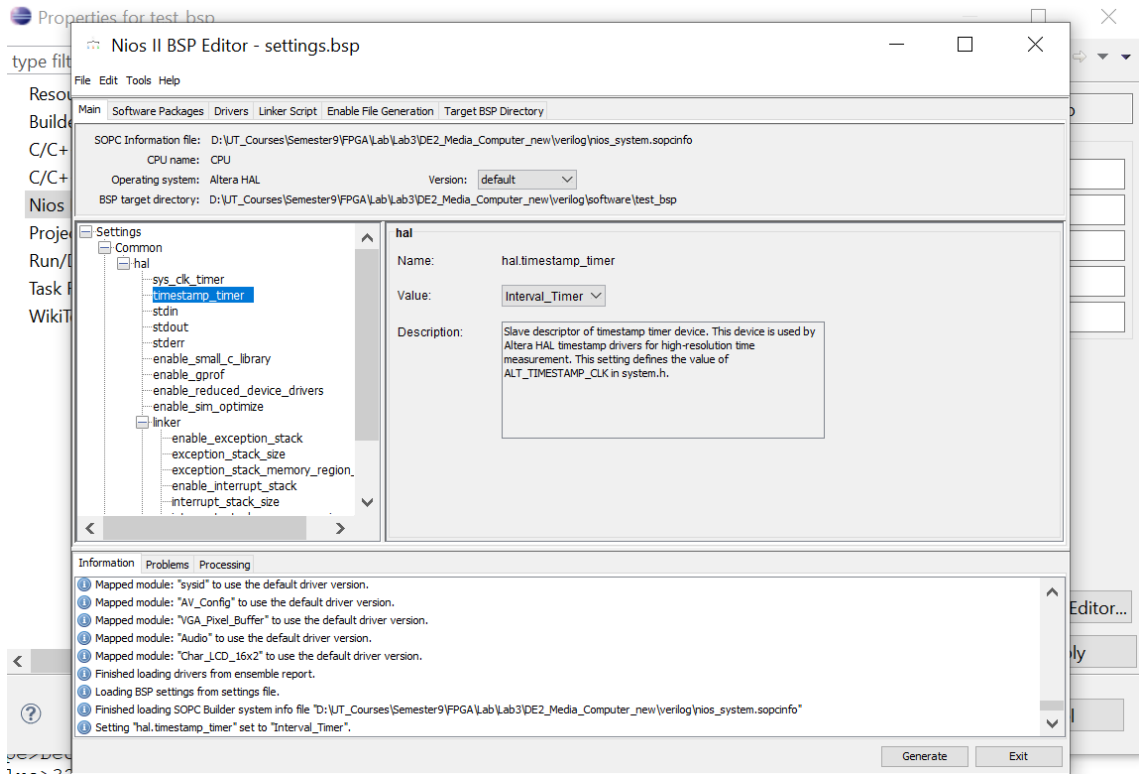
ابتدا روی پوشه ی BSP موجود در workspace کلیک کنید سپس مسیر زیر را طی کنید:

Project > Properties > NIOS II BSP Properties > BSP Editor… > timestamp_timer

در طی انجام این مراحل، مشابه دو تصویر زیر را مشاهده میکنید.



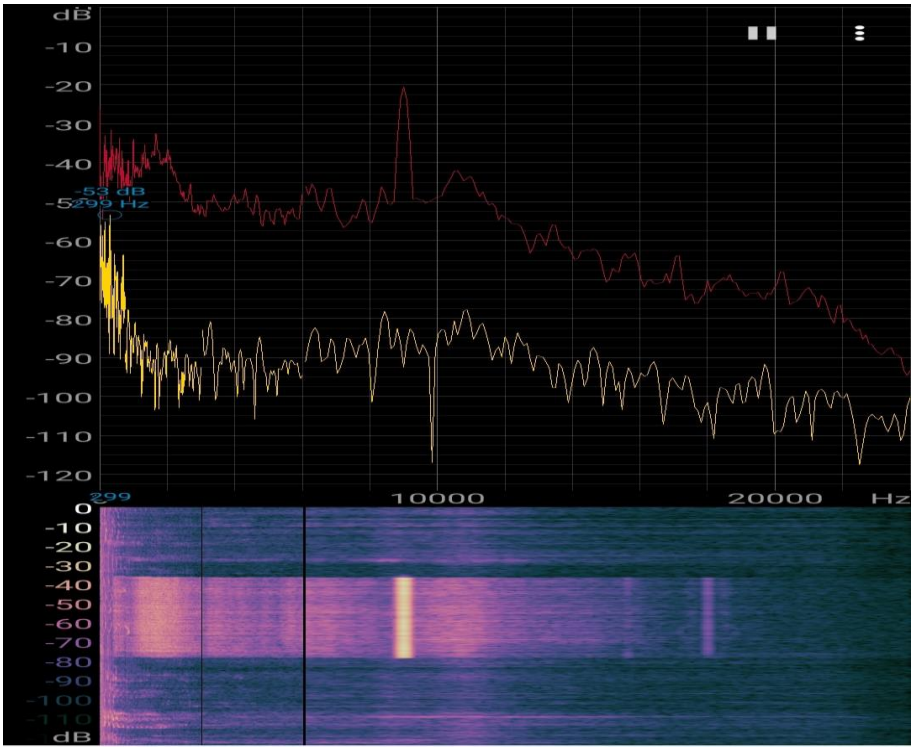
تصویر اول



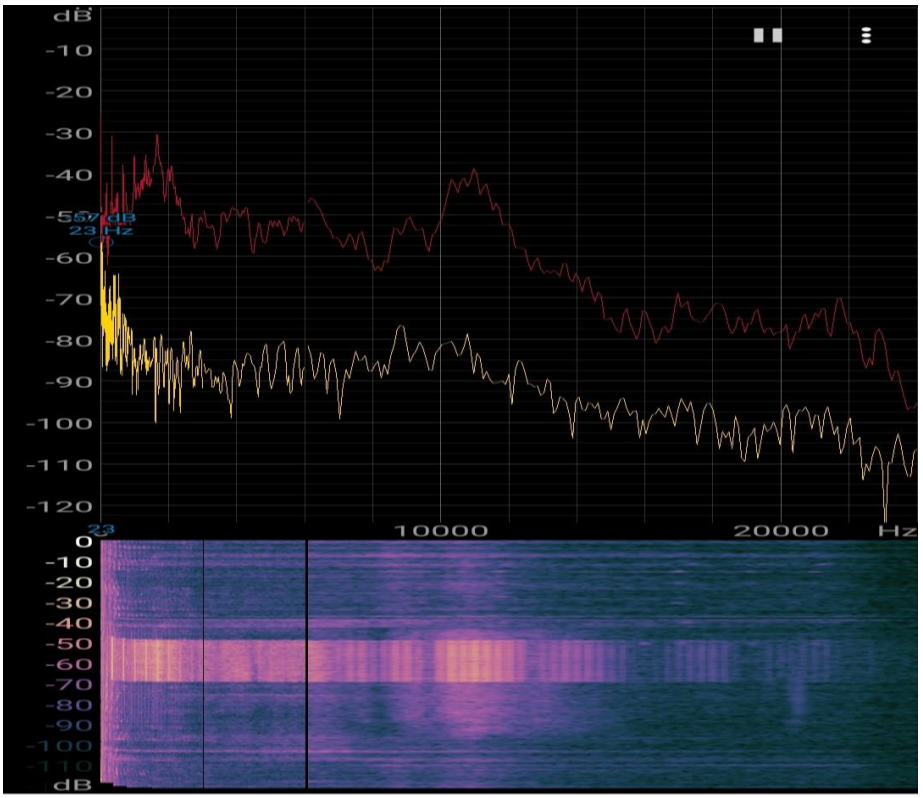
تصویر دوم

پس از طی مسیر گفته شده به تصویر دوم میرسیم، که در این مرحله باید در تنظیمات timestamp_timer، Value را به Interval_Timer تغییر دهیم تا بتوانیم از توابع مربوط به این کتابخانه بهره مند شویم. پس از انجام این تنظیمات، اورهای مربوط به استفاده از توابع این کتابخانه همگی رفع شده و میتوان اندازه گیری زمان را با دقت بالایی (به دلیل سخت افزاری بودن این فرآیند، در مقایسه با اندازه گیری نرم افزاری با Alarm) انجام داد.

فیلتراسیون نرم افزاری به درستی انجام شده که دو تصویر زیر نشان دهنده ی فرکانس صوت خروجی قبل و بعد از انجام فیلتراسیون است. پس از انجام فیلتراسیون، همانطور که در تصویری که در ادامه مشاهده میکنید، نویز موجود در فرکانس حدود 9kHz از بین میرود. نرم افزار مورد استفاده در این بخش Spectroid است که با استفاده از میکروفن موبایل، فرکانس اصوات محیط را مانند تصاویر زیر ثبت میکند:



خروجی صدا پیش از حذف نویز



خروجی اسپیکر پس از حذف نویز

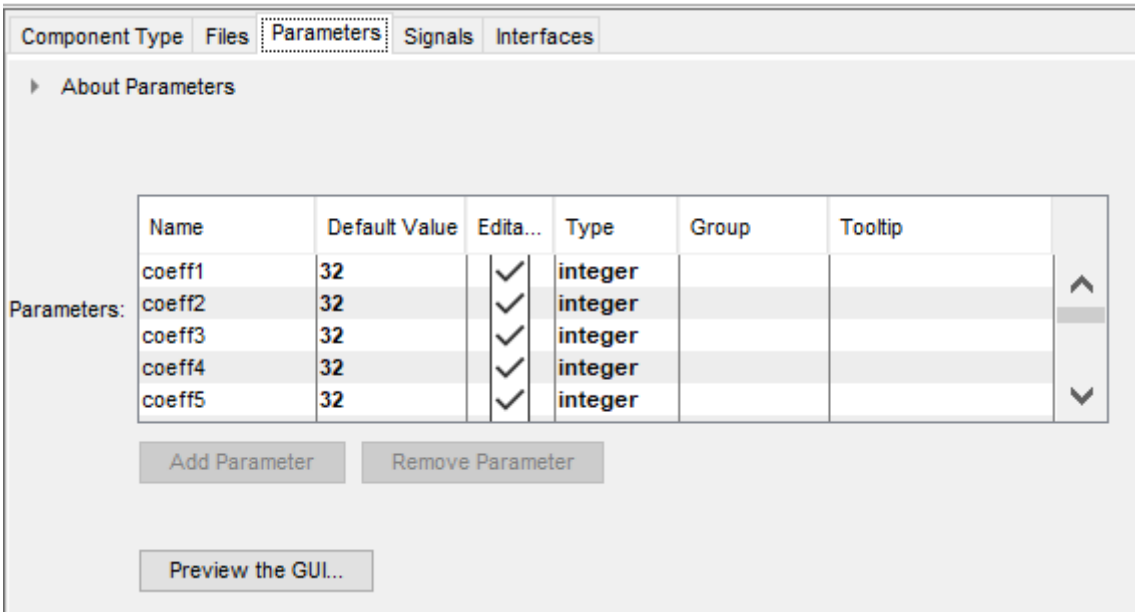
گام دوم

در این گام با اضافه کردن new component در Qsys یک custom instruction را به instruction های NIOS II اضافه میکنیم. مراحل انجام آن مانند همان مراحلی است که در اسلاید های درس به آنها اشاره شده. در این بخش به شرح تنظیمات خاص انجام این پروژه میپردازیم. در ابتدا برای آنکه ورودی 24 بیتی فایل تولید شده در متلب را به ورودی 32 بیتی تبدیل کنیم، ابتدای کد مربوط به فیلتر را به فرم زیر تغییر دادیم:

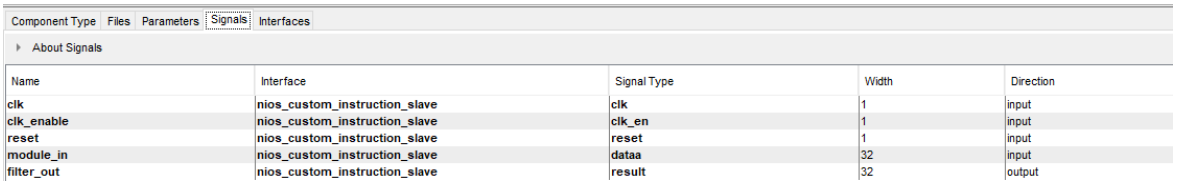
```
module filter_inputs48k_fully_serial
               (
                clk,
                clk_enable,
                reset,
                module_in,
                filter_out
                );

  input   clk;
  input   clk_enable;
  input   reset;
  input signed[31:0] module_in;
  output  signed [31:0] filter_out; //sfix32_En30

  wire signed [23:0] filter_in; //sfix24_En23

  assign filter_in=module_in[31:8];
```

خطوط هایلایت شده، کد هایی هستند که در فایل فیلتر تولید شده توسط متلب تغییر داده شده اند

در بخش parameters، ضرایب را که به طور پیش فرض 16 بیتی بودند را به ضرایب 32 بیتی تبدیل کردیم:



در بخش signals که بسیار در انجام تنظیمات این بخش اهمیت دارد، interface ها را به nios_custom_instruction_slave تغییر دادیم و ورودی و خروجی های مربوطه را با توجه به عملکردشان Signal Type مناسب را انتخاب کردیم که در تصویر زیر مشاهده میکنید:



سوال: نیازمندی های تعریف یک دستور Multicycle چیست؟

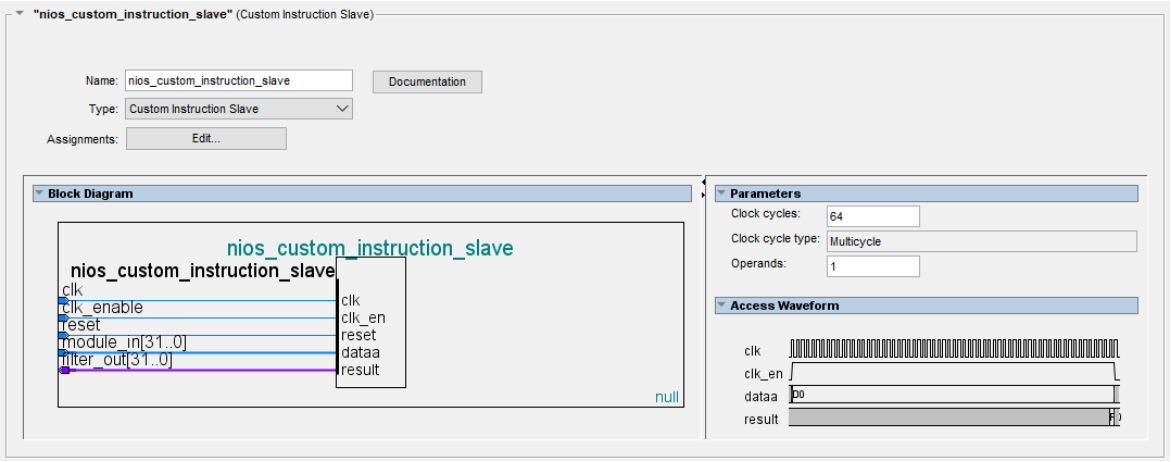
پاسخ: با توجه به توضیحات ارائه شده در فایل "Nios II Custom Instruction User Guide"، جهت تعریف دستور multicycle لازم است در ابتدا به این نکته توجه شود که آیا دستوری که میخواهیم اضافه کنیم در یک تعداد سیکل مشخص عملیات آن به پایان میرسد یا تعداد سیکل انجام عملیات آن متغیر است. اگر تعداد سیکل مشخص باشد در system generation تعداد سیکل مورد نیاز دستور را تعیین میکنیم. اما 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 در تصویر زیر آن ها را مشاهده میکنید:

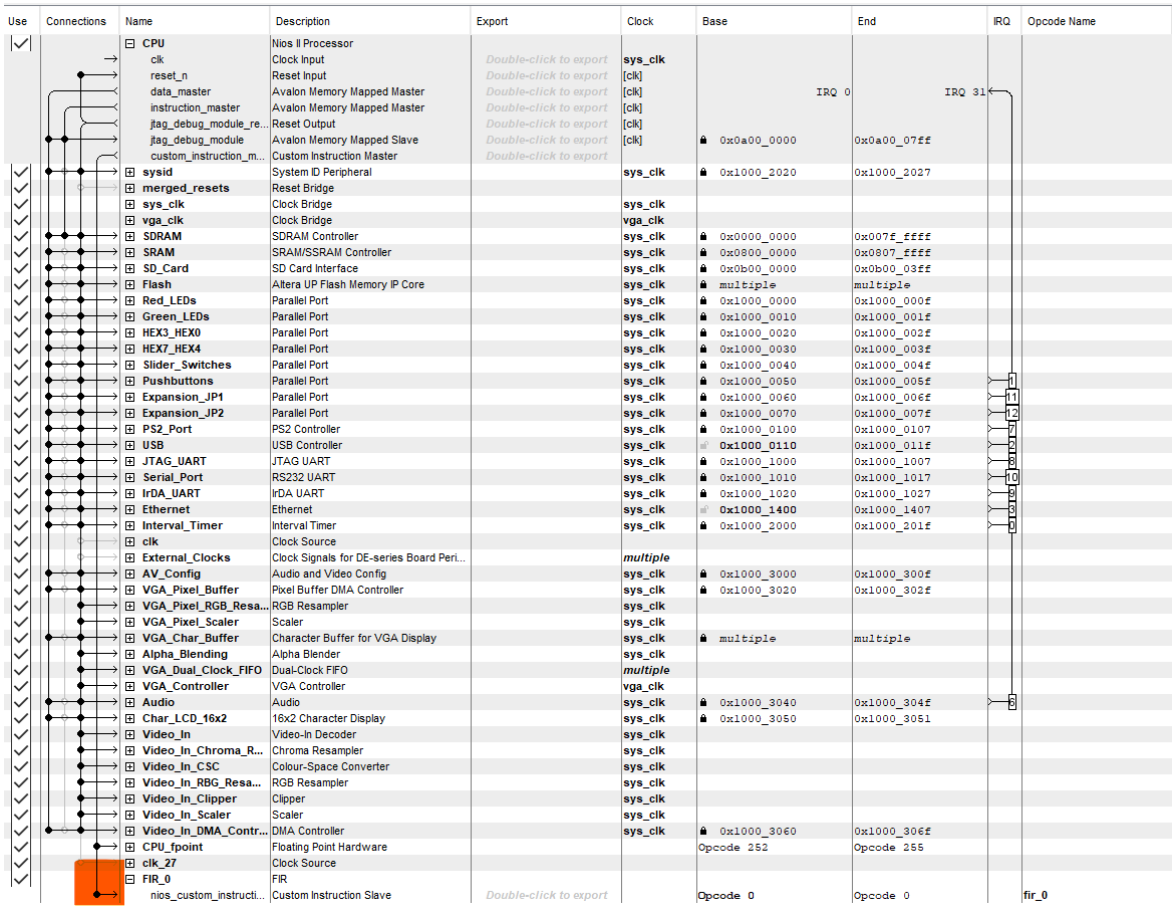
Port Name	Direction	Required	Description
clk	Input	Yes	System clock
clk_en	Input	Yes	Clock enable
reset	Input	Yes	Synchronous reset
start	Input	No	Commands custom instruction logic to start execution
done	Output	No	Custom instruction logic indicates to the processor that execution is complete
dataa[31:0]	Input	No	Input operand to custom instruction
datab[31:0]	Input	No	Input operand to custom instruction
result[31:0]	Output	No	Result of custom instruction

در میان پورت های بالا، clk, clk_en و reset پورت های اجباری برای ساخت multicycle custom instruction هستند و سایر پورت ها آپشنال هستند و برحسب نوع دستور میتوانیم آن ها را در بخش Signals برای نوع ورودی ها و خروجی های دستور تعیین کنیم.

در بخش Interfaces مانند تصویر زیر، تنظیمات لازم را اعمال میکنیم(با توجه به اینکه طول فیلتر 64 است تعداد سیکل ها را روی 64 تنظیم میکنیم و با توجه به اینکه یک ورودی داریم، Operands را برابر 1 قرار میدهیم):



در ادامه مانند تصویر زیر، custom instruction ساخته شده را به custom instruction master پردازنده NIOS II وصل میکنیم:



در پایان از تب Generation، فایل Qsys جدید را generate میکنیم و فایل کد سخت افزاری جدید به پروژه اضافه میشود.

گام سوم

همانطور که در دستور کار هم اشاره شده، ابتدا پروژه را در Quartus باز میکنیم و مجددا سنتز میکنیم. در تصویر زیر، میزان منابع استفاده شده توسط سیستم جدید را (پس از اضافه کردن دستور جدید) مشاهده میکنید:

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Mon Dec 20 19:08:48 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	DE2_Media_Computer
Top-level Entity Name	DE2_Media_Computer
Family	Cyclone II
Total logic elements	22,641
Total combinational functions	17,076
Dedicated logic registers	13,848
Total registers	13848
Total pins	416
Total virtual pins	0
Total memory bits	170,297
Embedded Multiplier 9-bit elements	24
Total PLLs	2

برای مقایسه منابع استفاده شده با اضافه کردن دستور جدید با میزان منابع مورد استفاده لازم است ابتدا تصویری از میزان منابع مورد استفاده پیش از اضافه کردن دستور داشته باشیم و آن را با پس از اضافه کردن دستور مقایسه کنیم:

Flow Summary	
Flow Status	Successful - Wed Dec 22 21:29:23 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	DE2_Media_Computer
Top-level Entity Name	DE2_Media_Computer
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	19,897
Total combinational functions	15,901
Dedicated logic registers	12,201
Total registers	12201
Total pins	416
Total virtual pins	0
Total memory bits	170,297
Embedded Multiplier 9-bit elements	21
Total PLLs	2

میزان منابع مورد استفاده پیش از اضافه کردن دستور اختصاصی جدید

از مقایسه ی دو تصویر بالا، میزان منابع اضافه شده به ترتیب زیر خواهد بود:

Total logic elements	+2744
Total registers	+1647
Embedded Multiplier 9-bit elements	+3

میزان منابع اضافه مورد استفاده پس از اضافه کردن دستور جدید تقریبا با میزان منابع مورد استفاده گزارش شده flow summary مربوط به فیلتر سریال برابر است(لاجیک المنت ها بیشتر است، رجیستر ها کمی بیشتر و تعداد ضرب کننده های اشاره شده، یک واحد کمتر است).

به منظور استفاده از دستور جدید، با ساخت BSP جدید با فایل sopcinfo. ساخته شده، custom instruction ای که تعریف کردیم به پروژه اضافه میشود و توابع آن را میتوان در system.h مشاهده کرد. در تصویر زیر، ماکروهای custom instruction مربوط به فیلتر FIR ای که در Qsys اضافه کردیم را میتوانید مشاهده کنید:

```
/*
 * Custom instruction macros
 *
 */

#define ALT_CI_FIR_0(A) __builtin_custom_ini(ALT_CI_FIR_0_N,(A))
#define ALT_CI_FIR_0_N 0x0
#define ALT_CI_FPOINT_0(n,A,B) __builtin_custom_inii(ALT_CI_FPOINT_0_N+(n&ALT_CI_FPOINT_0_N_MASK),(A),(B))
#define ALT_CI_FPOINT_0_N 0xfc
#define ALT_CI_FPOINT_0_N_MASK ((1<<2)-1)
```

همانطور که میتوان حدس زد، تابع تک ورودی ALT_CI_FIR_0 همان فیلتر ماست که به عنوان ورودی، تک داده ی صوت (32 بیتی) را میگیرد و تک داده ی صوت فیلتر شده را (32 بیتی) تحویل میدهد. با استفاده از این تابع کد انجام محاسبات فیلتر (make_echo) را مطابق شکل زیر تغییر میدهیم:

```
void make_echo(){
    //int i,j;
    //double accumulator_l, accumulator_r;
    /*for(i=5000;i<BUF_SIZE;i++){
        l_buf_echo[i]=l_buf[i-5000]+l_buf[i-1000];
        r_buf_echo[i]=r_buf[i-5000]+r_buf[i-1000];
    }*/
    int i;
    for(i=0;i<BUF_SIZE;i++)//with custom instruction
        l_buf_echo[i]=ALT_CI_FIR_0(l_buf[i]);
    for(i=0;i<BUF_SIZE;i++)
        r_buf_echo[i]=ALT_CI_FIR_0(r_buf[i]);
    /*for(i=0;i<BUF_SIZE;i++){//software FIR computation
        printf("%d\n",i);
        accumulator_l=0;
        accumulator_r=0;
        for(j=i;j>=0 && i-j<64;j--){
            accumulator_l=accumulator_l+(double)(l_buf[j]>>8)*coeffs[i-j];
            accumulator_r=accumulator_r+(double)(r_buf[j]>>8)*coeffs[i-j];
        }
        l_buf_echo[i]=((unsigned int)(accumulator_l));
        r_buf_echo[i]=((unsigned int)(accumulator_r));
    }*/
    return;
}
```

اگر به بخشی که کامنت نیست توجه کنید، دو حلقه ی جداگانه به منظور انجام محاسبات روی صوت ضبط شده در بافر چپ و بافر راست (اسپیکر چپ و راست به منظور پخش استریو صدا) وجود دارد. با دادن تک تک داده های این دو بافر به فیلتر با استفاده از دستور custom instruction ساخته شده، میتوان عملیات فیلتراسیون را انجام داد.

نکته ی بسیار جالب در استفاده از این روش اینست که زمان انجام عملیات آن بسیار سریع بوده و در حدود 2 تا 3 ثانیه است.

نتیجه گیری

اجرای دستوراتی که نیاز به محاسبات ویژه ای دارند، اگر به صورت نرم افزاری پیاده سازی شود، چون پردازنده ی مورد استفاده ی آن همان پردازنده ی کلی مدار است، به دلیل آنکه ممکن است آن پردازنده برای انجام محاسبات آن دستور بهینه نباشد (چون پردازنده ی general purpose است) ممکن است انجام عملیات آن دستور به صورت نرم افزاری، علاوه بر سربار نرم افزاری ای که نرم افزار روی سخت افزار دارد(به جهت اجرای واسطه ای سیستم عامل به کار برده شده در آن دستگاه)، انجام محاسبات آن نیز بسیار طول بکشد.

اما در programmable gate array چون میتوانیم سخت افزار را تا حدی خودمان config کنیم و از این طریق custom instruction ایجاد کنیم، میتوانیم با اضافه کردن دستور اختصاصی برای انجام عملیات مورد نظر، آن عملیات را به صورت سخت افزاری با کانفیگ سخت افزاری مخصوص اجرای آن دستور، با سرعت بسیار بالاتری در مقایسه با روش نرم افزاری انجام دهیم.

در این آزمایش هم مشاهده کردیم که اجرای نرم افزاری فیلتر پایین گذر، چند ده دقیقه به طول انجامید در حالی که اجرای آن با استفاده از دستور اختصاصی، در چند ثانیه به اتمام رسید.

مراجع

1) Nios II Custom Instruction User Guide

2) Developing Programs Using the Hardware Abstraction Layer