به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

مدارهای مبتنی بر FPGA

آزمایش شماره 4

علیرضا جابری راد

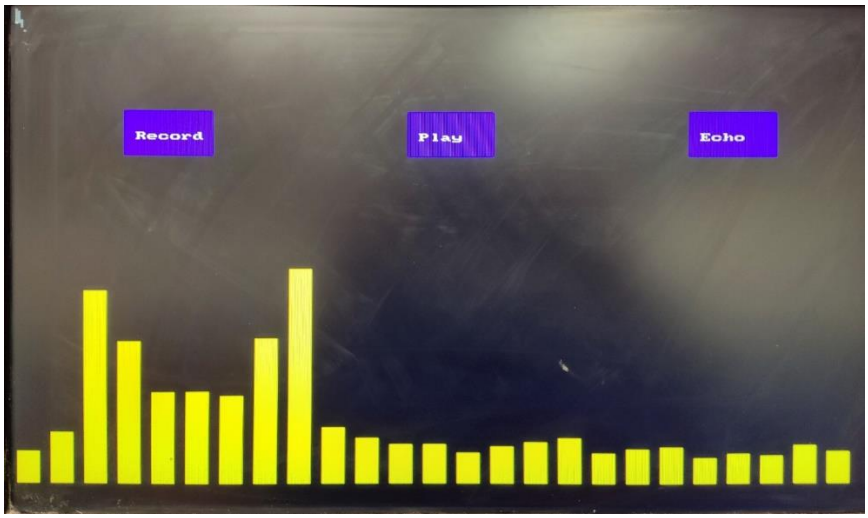
نیما سلیمی

زمستان 1400

فهرست

چکیده

در این آزمایش قصد داریم یک Audio Player با قابلیت ضبط و پخش صدا طراحی کنیم. در این Audio Player پس از ضبط صدا، میانگین دامنه صوت ضبط شده در بازه های مختلف محاسبه شده و با استفاده از یکسری مستطیل مشخص میشوند. در تصویر زیر بار های زرد رنگ نشان دهنده ی دامنه‌ی صوت ضبط شده در 25 بازه ی مختلف است:



همچنین حین پخش صدا، هر بازه ای که در حال پخش است، با استفاده از یک مستطیل در زیر نمودار دامنه صوت، نمایش داده میشود. در تصویر زیر مربع های آبی رنگ در حین پخش صوت نمایش داده میشوند:

در این آزمایش پیاده سازی محاسبه گر دامنه صوت، در دو بخش سخت افزاری و نرم افزاری انجام شده که در ادامه نحوه ی پیاده سازی و عملکرد آنها را با هم مقایسه میکنیم.

طراحی سیستم به صورت نرم افزاری

در این بخش، برای اینترفیس ضبط و پخش صدا از همان کد آزمایش قبلی استفاده کردیم. در آزمایش های قبلی پس از ضبط صدا make_echo_flag یک شده و محاسبات مربوط به تولید صدای اکو شده شروع میشد. در این آزمایش هم از همان فلگ استفاده کردیم و با یک شده آن، محاسبات مربوط به متوسط دامنه صوت ضبط شده شروع میشود. تابع calc_avg جهت محاسبه‌ی متوسط دامنه صوت ضبط شده مورد استفاده قرار میگیرد. تصویر زیر بدنه ی این کد را نشان میدهد:

```
void calc_avg(){
    int i,j;
    int size=BUF_SIZE/N;
    unsigned int temp;
    for(i=0;i<N;i++){
        avg[i]=0;
        for(j=0;j<size;j++){
            temp= ((l_buf[i*size+j] & 0x80000000) == 0x80000000) ? ((l_buf[i*size+j] ^ 0xffffffff)+1) : l_buf[i*size+j];
            avg[i]=avg[i]+(unsigned long long) temp;
        }
    }
    return;
}
```

جهت کوتاه تر شدن زمان انجام محاسبات و چون میکروفونی که در آزمایشگاه داریم **mono** است(و صوت ذخیره شده در هر دو بافر یکسان است)، فقط یک بافر را ملاک محاسبات قرار دادیم. مقدار متوسط دامنه هر **portion** در بخش نرم افزاری، در آرایه ی 64 بیتی **avg** ذخیره میشود (البته در این تابع تنها مجموع خانه های آرایه محاسبه میشود). صوتی که توسط میکروفون به بورد انتقال میابد و در بافر ذخیره میشود، به صورت **signed** ذخیره میشود (با وجود آنکه در کد **C** به صورت **unsigned** تعریف شده) و برای آنکه **absolute value** آن بدست آید بایستی **two's complement** آنرا محاسبه کنیم. متغیر **temp** جهت محاسبه ی قدر مطلق هر یک از خانه های بافر مورد استفاده قرار گرفته است.

سپس جهت نمایش مستطیل های نمایش دهنده‌ی متوسط دامنه تابع plot_audio را طراحی و پیاده سازی کردیم که کد آنرا در تصویر زیر مشاهده میفرمایید:

```
void plot_audio(alt_up_pixel_buffer_dma_dev* pixel_buffer_dev){
    short color;
    int i;
    unsigned int size=BUF_SIZE/N;
    int scale=0;
    unsigned long long int max_avg=0;

    for(i=0;i<N;i++){//SUM -> AVG
        hw_avg[i]=hw_avg[i]/size;
    }

    for(i=0;i<N;i++){
        avg[i]=avg[i]/size;
        printf("Sofware Calculation Result: %llu\nHardware Calculation Result: %llu\n",avg[i],hw_avg[i]);
    }

    for(i=0;i<N;i++){
        if(max_avg<hw_avg[i])
            max_avg=hw_avg[i];
    }

    scale = max_avg/100;
    color = 0xff00;
    for(i = 0; i < N; i++){
        alt_up_pixel_buffer_dma_draw_box (pixel_buffer_dev, i * col_size + 2, 225, ((i+1) * col_size)-3 ,(225 - (hw_avg[i] / scale)), color, 0);
        }
    return;
}
```

در این بخش ابتدا میانگین هر یک از مجموع های حساب شده در تابع **calc_avg** و تابع شتابگر سخت افزاری محاسبه میشود. سپس حداکثر متوسط دامنه از میان **N** بخش بافر محاسبه میشود. در ادامه هر یک از مقادیر بر حسب ماکسیمم مقدار **scale** میشوند و با استفاده از تابع **alt_up_pixel_buffer_dma_draw_box** به صورت بار های مستطیل شکل نمایش داده میشوند.

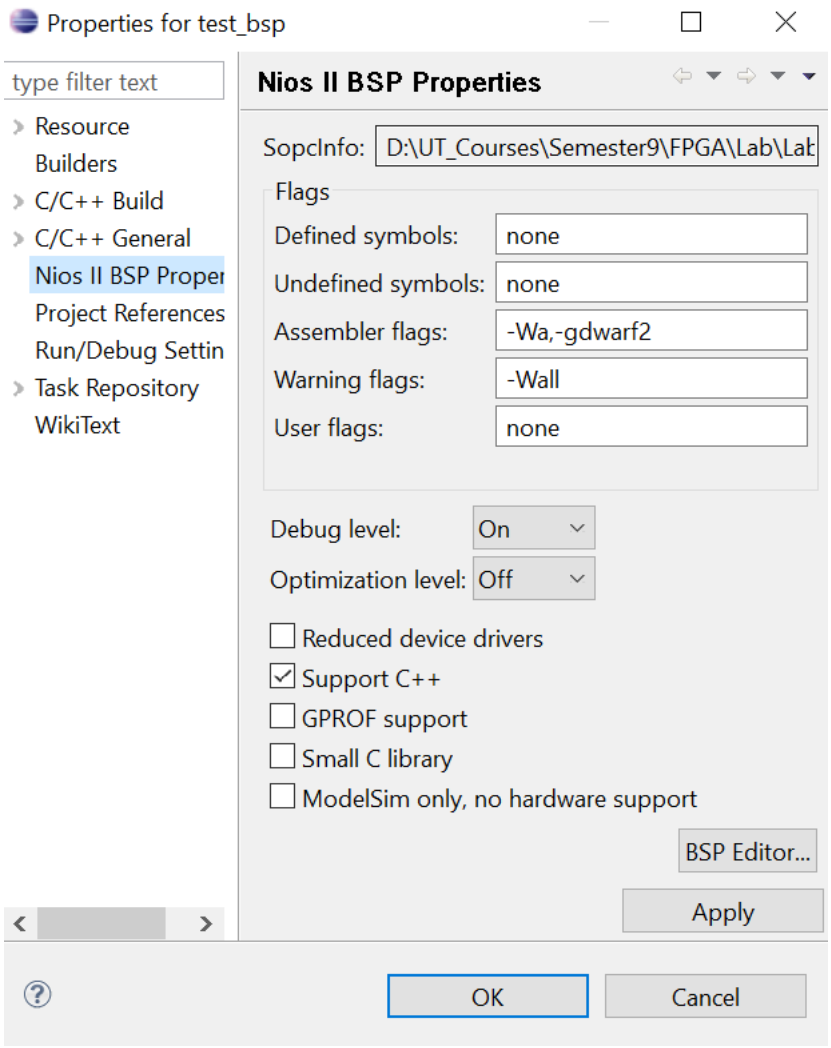
پس از رسم نمودار مربوط به متوسط دامنه، فلگ make_echo_flag صفر میشود و فرآیند ضبط صدا و رسم نمودار ها پایان می‌یابد.

اندازه گیری زمان انجام محاسبات نرم افزاری

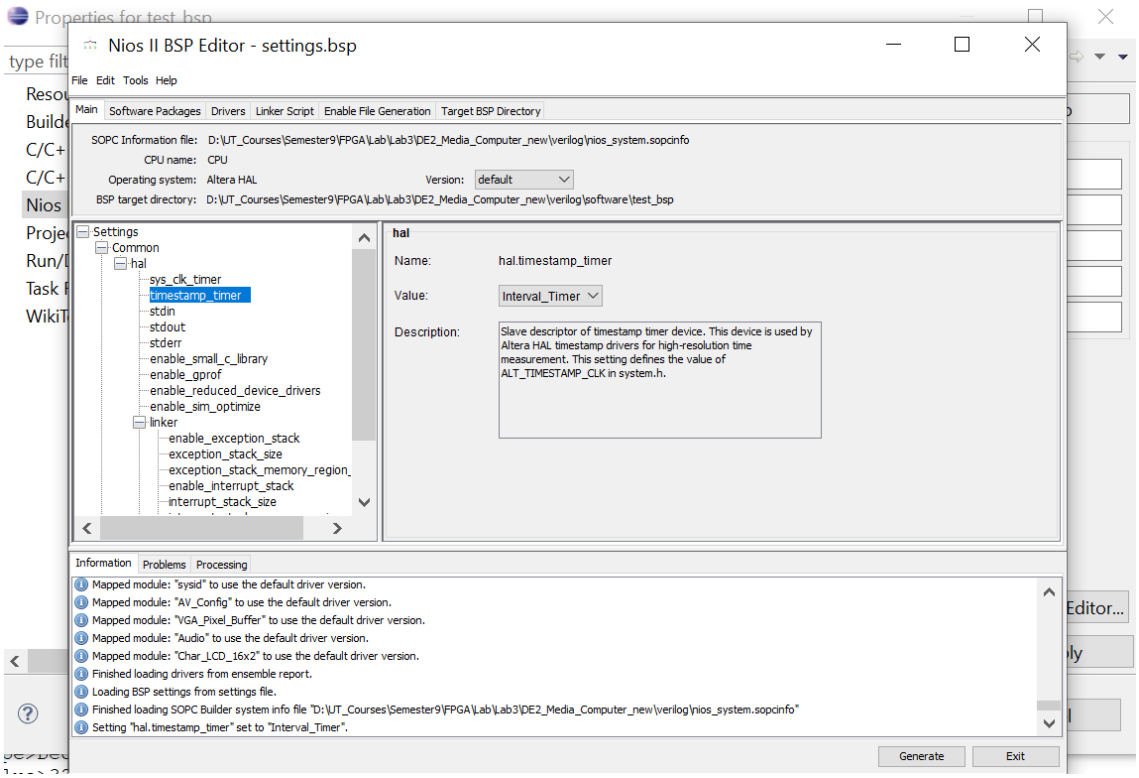
جهت اندازه گیری زمان انجام محاسبات از timestamp_timer بهره بردیم. جهت استفاده از timestamp_timer و توابع آن لازم است ابتدا یکسری تنظیمات و کتابخانه ها به فایل پروژه اضافه شوند. بدین منظور ابتدا ابتدا روی پوشه ی BSP موجود در workspace کلیک کنید سپس مسیر زیر را طی کنید:

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. همچنین دقت کنید که حتما در sys_clk_timer، Value را به None تغییر دهید تا دسترسی کلاک تایمر فقط در اختیار timestamp_timer باشد و استفاده همزمان این دو کامپوننت منجر به اختلال عملکردشان نشود. در صورتی که دسترسی به کلاک Interval_Timer را از sys_clk_timer نگیریم، تابع alt_timestamp_freq() به درستی کار نخواهد کرد و فرکانس کلاک Interval Timer را نمیتواند گزارش کند.

در ادامه لازم است جهت بهره گیری از توابع مربوط به timestamp_timer کتابخانه alt_timestamp.h را با استفاده از دستور #include “sys/alt_timestamp.h” به پروژه اضافه کنیم. در تصویر زیر بخشی از کد که زمان اجرای کد نرم افزاری را اندازه گیری میکند را مشاهده میکنید:

```
//alt_timestamp_start();
timestamp_freq=alt_timestamp_freq();
if(timestamp_freq == 0)
    printf("timestamp hardware not working\n");
printf("Calculation Started...\n");

//Software average calculations
alt_timestamp_start();
calc_avg();
printf("Software Calculation Finished in %.3f seconds\n", (float)alt_timestamp()/(float)timestamp_freq);
```

در کد بالا، alt_timestamp_freq فرکانس کلاک Interval Timer را برمیگرداند. alt_timestamp_start شمارنده ی مربوط به timestamp_timer را صفر کرده و شمارش بر حسب کلاک Interval Timer شروع میشود. با استفاده از دستور alt_timestamp آخرین مقدار ذخیره شده در شمارنده برگردانده میشود و با تقسیم این مقدار بر فرکانس کلاک، زمان طی شده برای انجام محاسبات بدست می‌آید. در تصویر زیر نتیجه ی اجرای این کد را مشاهده میکنید که در آن انجام محاسبات نرم افزاری حدود 3.378 ثانیه به طول انجامیده است:

```
Calculation Started...
Software Calculation Finished in 3.378 seconds
```

طراحی شتاب‌دهنده سخت افزاری

طراحی رابط Avalon Memory-Mapped Slave:

عملکرد این بخش وابسته به مقادیر موجود در رجیسترهای این رابط است که با توجه به تصویر زیر بایستی در نظر گرفته شوند:

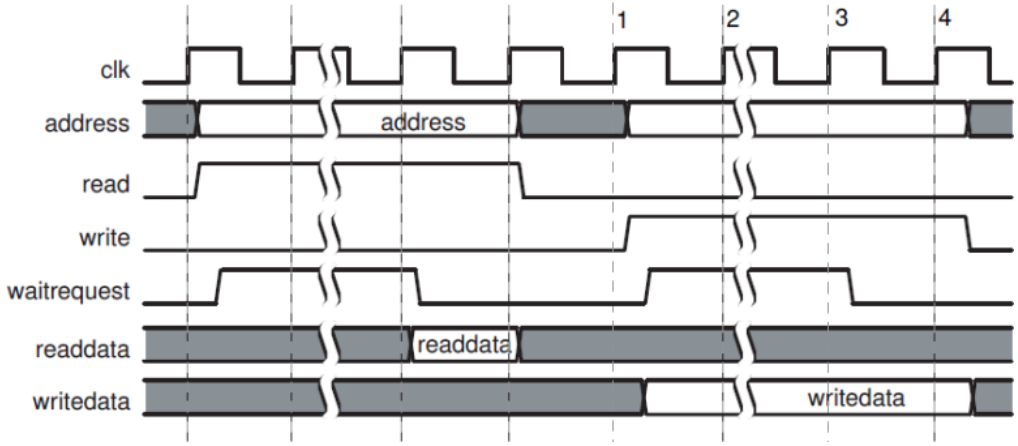
Slave Address	31	30..12	11..1	0	
00	Done	Size	Num	Go	Config. Reg.
01					Right Addr.
10					Left Addr.
11					Out Addr.

لذا در کد این بخش هم از 4 رجیستر اصلی بدین منظور استفاده شده. عملکرد کنترلی اصلی به ماژول accelerator منتقل کردم و عملکردی که در این ماژول لحاظ شده است، تنها نوشتن و خواندن از این چهار رجیستر است که در کد زیر آنرا مشاهده میفرمایید:(نکته: این رابط بدون سیگنال wait request طراحی شده و مقدار آن در این دیزاین همواره صفر است)

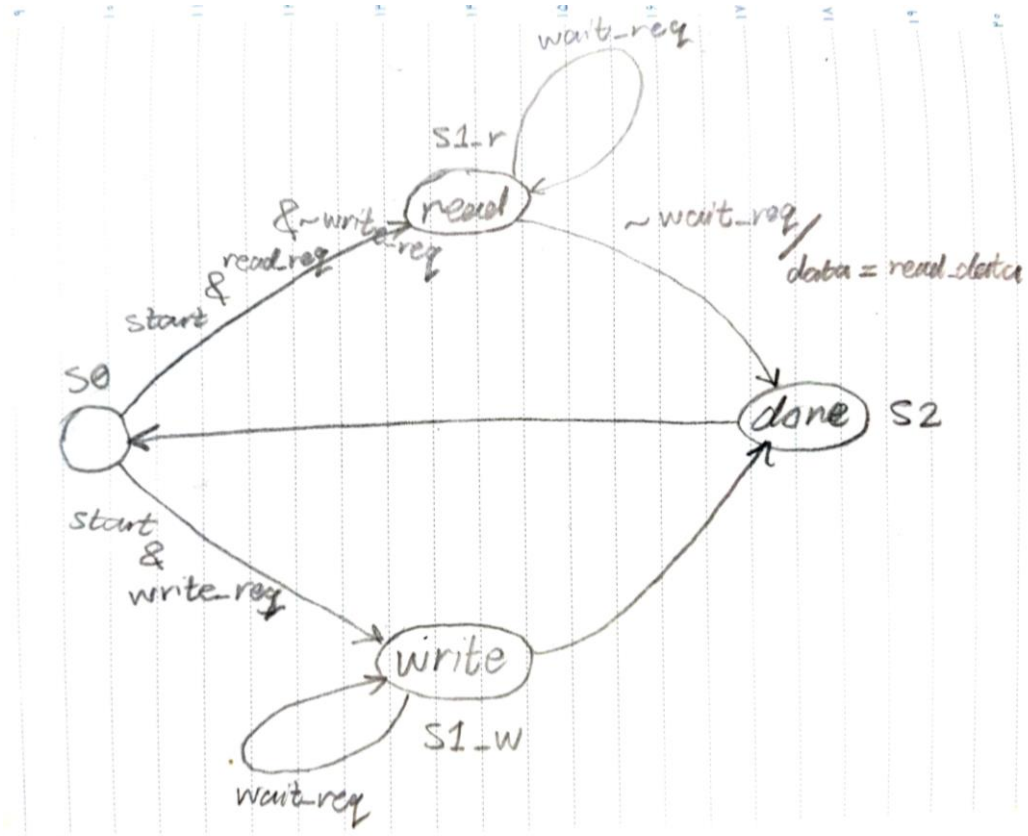
```
always @(posedge CSI_CLOCK_CLK, posedge CSI_CLOCK_RESET)
begin
  if(CSI_CLOCK_RESET == 1)
  begin
    slv_reg0 <= 0;
    slv_reg1 <= 0;
    slv_reg2 <= 0;
    slv_reg3 <= 0;
  end
  else  begin
    slv_reg0[31] <= DONE;
    if(AVS_AVALONSLAVE_WRITE) begin
      // address is bytewise, it must be devided by 4
      case(AVS_AVALONSLAVE_ADDRESS >> 2)
      0: slv_reg0[30:0] <= AVS_AVALONSLAVE_WRITEDATA[30:0];
      1: slv_reg1 <= AVS_AVALONSLAVE_WRITEDATA;
      2: slv_reg2 <= AVS_AVALONSLAVE_WRITEDATA;
      3: slv_reg3 <= AVS_AVALONSLAVE_WRITEDATA;
      default:
      begin
        slv_reg0[30:0] <= slv_reg0[30:0];
        slv_reg1 <= slv_reg1;
        slv_reg2 <= slv_reg2;
        slv_reg3 <= slv_reg3;
      end
      endcase
    end
  end
end

always @(*) begin
  case(AVS_AVALONSLAVE_ADDRESS >> 2)
    0:  read_data = slv_reg0;
    1:  read_data = slv_reg1;
    2:  read_data = slv_reg2;
    3:  read_data = slv_reg3;
    default :  read_data = {AVS_AVALONSLAVE_DATA_WIDTH{1'b0}};
  endcase
end
```

طراحی رابط Avalon Memory-Mapped Master

طراحی این بخش بر مبنای شکل موج زیر انجام شده است:



این بخش صرفا واسط عملکرد خواندن و نوشتن بین شتاب دهنده و SDRAM خواهد بود که عملکرد کنترلی آن طبق استیت ماشین زیر طراحی شده است:



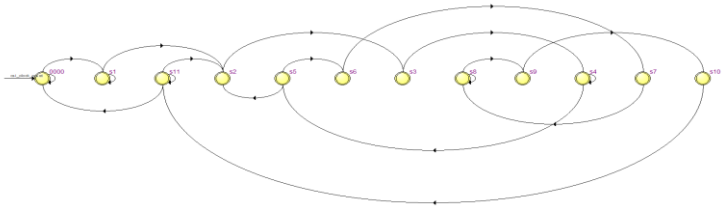
کد زیر متناظر با استیت ماشین طراحی شده است:

```
always @(posedge CSI_CLOCK_CLK, posedge CSI_CLOCK_RESET) begin
  if(CSI_CLOCK_RESET == 1) begin
    state <= s0;
    data <= {AVM_AVALONMASTER_DATA_WIDTH{1'b0}};
  end
  else begin
    case(state)
      s0 : begin
        if(START) begin
          if(write_req)
            state <= s1_w;
          else if(read_req)
            state <= s1_r;
          else
            state <= s0;
        end
      end

      s1_r : begin
        if(AVM_AVALONMASTER_WAITREQUEST == 1'b1)
          state <= s1_r;
        else begin
          data <= AVM_AVALONMASTER_READDATA;
          state <= s2;
        end
      end

      s1_w : begin
        if(AVM_AVALONMASTER_WAITREQUEST == 1'b1)
          state <= s1_w;
        else
          state <= s2;
      end

      s2 : state <= s0;
    endcase
  end
end

always @(*) begin
  read = 1'b0;
  write = 1'b0;
  done = 1'b0;
  case(state)
    s1_r : read = 1'b1;
    s1_w : write = 1'b1;
    s2 : done = 1'b1;
  endcase
end
```

طراحی مدار محاسبه دامنه

در این بخش، ماژول اصلی شتاب دهنده را طراحی میکنیم که قرار است عملکرد جمع به صورت سخت افزاری انجام شود. ابتدا ماژول های طراحی شده در دو بخش قبل را در این ماژول اینستنس میگیریم تا یک ساختار واحد در درون شتاب دهنده، شامل محاسبه گر مجموع دامنه، slave و master تشکیل شود.

گام اصلی در طراحی این مدار، دقت در تعامل شتابدهنده با پردازنده و SDRAM با واسطه ی Avalon Bus است. بخش اصلی این مدار، بخش کنترل کننده ی آن است که به دلیل طولانی بودن طراحی استیت ماشین آن، در فایل جداگانه ای در پوشه‌ی Report فایل پی دی اف accelerator state machine را قرار دادم. جهت داشتن یک دید کلی نسبت به استیت ماشین طراحی شده، تصویر زیر که از محیط کوارتوس عکس برداری شده است را مشاهده میفرمایید:



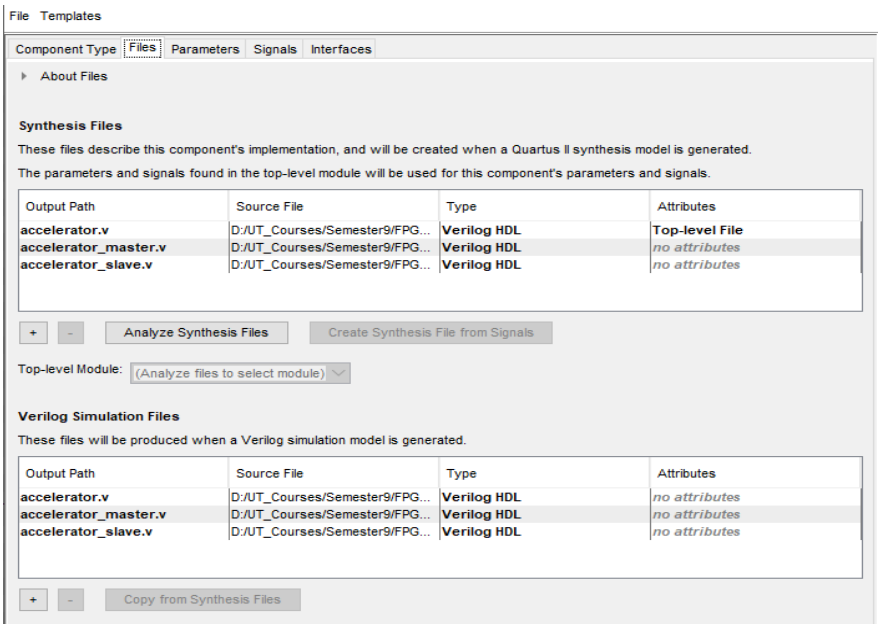
	Source State	Destination State	Condition
1	0000	0000	(!AVS_AVALONSLAVE:AVS_AVALONSLAVE_INST)
2	0000	s1	(AVS_AVALONSLAVE:AVS_AVALONSLAVE_INST)
3	s1	s1	(AVS_AVALONSLAVE:AVS_AVALONSLAVE_INST)
4	s1	s2	(!AVS_AVALONSLAVE:AVS_AVALONSLAVE_INST)
5	s2	s3	
6	s3	s4	
7	s4	s4	(!AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
8	s4	s5	(AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
9	s5	s2	(LessThan0)
10	s5	s6	(!LessThan0)
11	s6	s7	
12	s7	s8	
13	s8	s9	(AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
14	s8	s8	(!AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
15	s9	s10	
16	s10	s11	
17	s11	0000	(!LessThan1) (AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
18	s11	s2	(LessThan1) (AVM_AVALONMASTER_MAGNITUDE:AVM_AV...
19	s11	s11	(!AVM_AVALONMASTER_MAGNITUDE:AVM_AV...

کد مربوط به کنترلر بالا، حجم تقریبا زیادی دارد که برای آنکه بتوانید دید مناسبی را به آن پیدا کنید، توصیه میکنم به فایل پی دی اف accelerator state machine مراجعه فرمایید. در ادامه کد نوشته شده را سنتز میکنم و میزان مصرف منابع سخت افزاری را در آن مشاهده میکنیم:

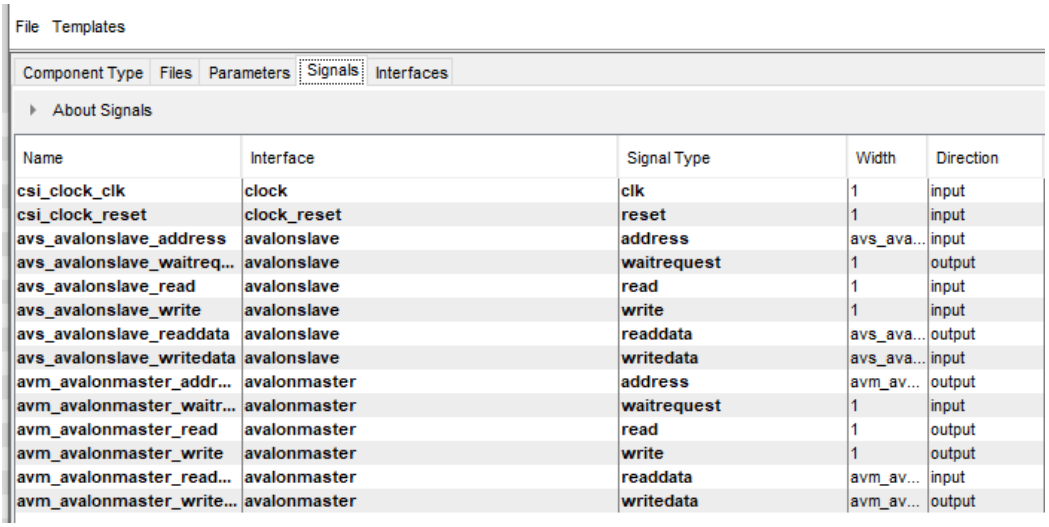
Flow Summary	
Flow Status	Successful - Mon Jan 24 20:38:44 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	accelerator
Top-level Entity Name	accelerator
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	507 / 33,216 (2 %)
Total combinational functions	408 / 33,216 (1 %)
Dedicated logic registers	460 / 33,216 (1 %)
Total registers	460
Total pins	172 / 475 (36 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

حال مدار طراحی شده را به پروژه آزمایش گذشته اضافه میکنیم و کانفیگ سخت افزاری آنرا در محیط Qsys به ترتیبی که در ادامه گفته میشود، انجام میدهیم.

ابتدا از تب tools وارد محیط Qsys میشویم. سپس با دبل کلیک روی New Component یک کامپوننت جدید اضافه میکنیم. پس از انتخاب نام سخت افزاری که میخواهیم اضافه کنیم، تب files را انتخاب کرده و فایل های وریلاگ طراحی شده در این بخش را به پروژه می‌افزاییم و پس از کلیک بر روی Analyze Synthesis Files و پس از آن کلیک بر روی Copy from Synthesis Files فرآیند این تب به پایان میرسد و تصویری مانند تصویر زیر را خواهیم داشت و میتوانیم ادامه فرآیند را پیگیری کنیم:



در ادامه در تب signals تنها نکته ای که در طراحی من متفاوت بود و لازم بود دقت کنم این بود که ریست من از نوع active high است و در این صورت کانفیگ signals باید به صورت زیر باشد و چون اسامی سیگنال ها فرم استاندارد دارد، لازم نیست چیزی تغییر کند و همه سیگنال ها به درستی ست شده اند:

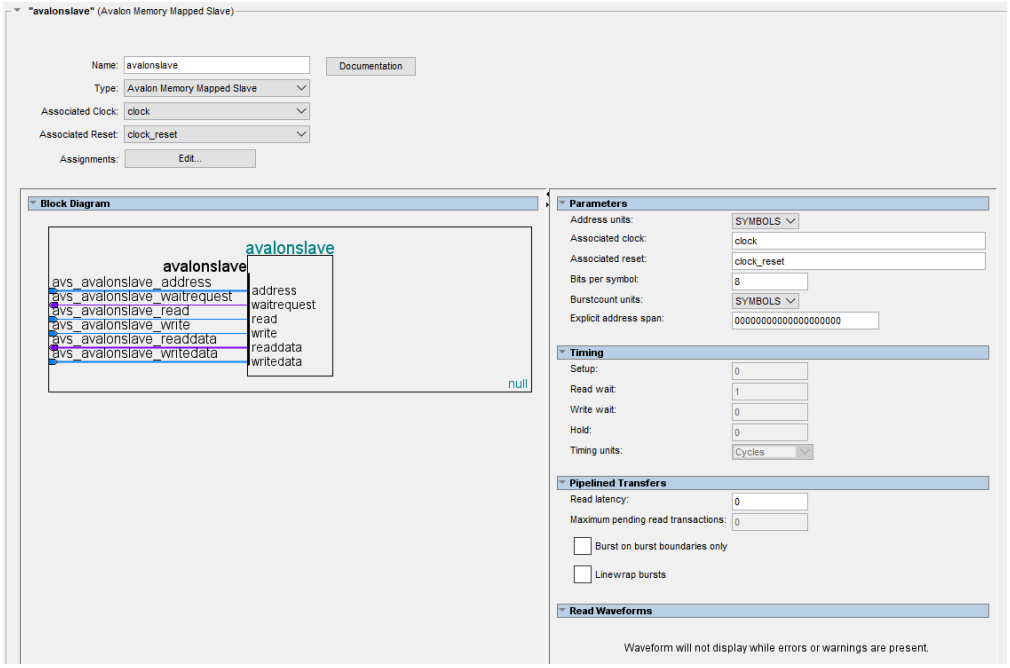

File Templates

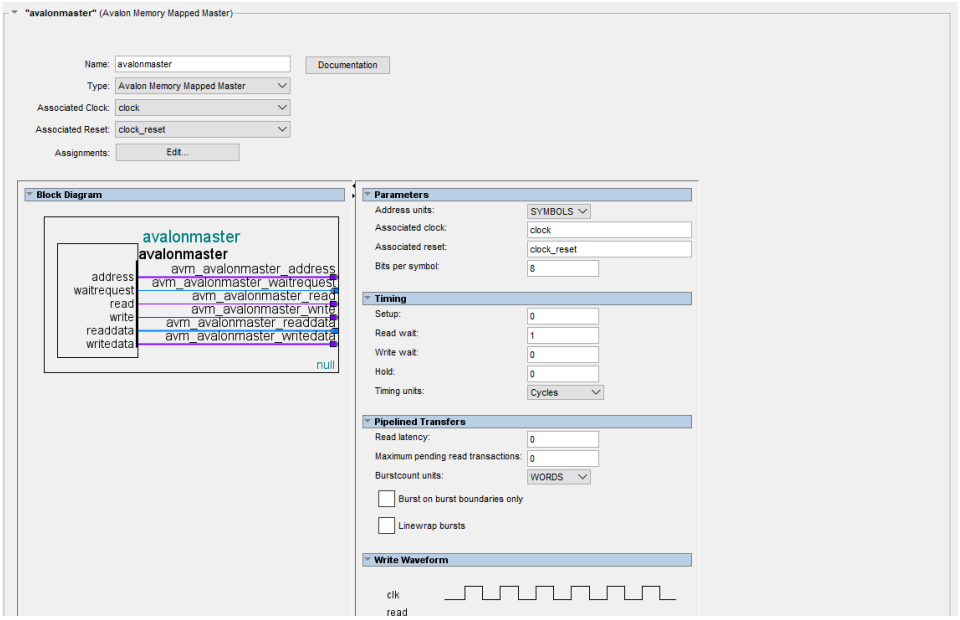
Component Type | Files | Parameters | Signals | Interfaces

About Signals

Name	Interface	Signal Type	Width	Direction
csi_clock_clk	clock	clk	1	input
csi_clock_reset	clock_reset	reset	1	input
avs_avalonslave_address	avalonslave	address	avs_ava...	input
avs_avalonslave_waitreq...	avalonslave	waitrequest	1	output
avs_avalonslave_read	avalonslave	read	1	input
avs_avalonslave_write	avalonslave	write	1	input
avs_avalonslave_readdata	avalonslave	readdata	avs_ava...	output
avs_avalonslave_writedata	avalonslave	writedata	avs_ava...	input
avm_avalonmaster_addr...	avalonmaster	address	avm_av...	output
avm_avalonmaster_waitr...	avalonmaster	waitrequest	1	input
avm_avalonmaster_read	avalonmaster	read	1	output
avm_avalonmaster_write	avalonmaster	write	1	output
avm_avalonmaster_read...	avalonmaster	readdata	avm_av...	input
avm_avalonmaster_write...	avalonmaster	writedata	avm_av...	output

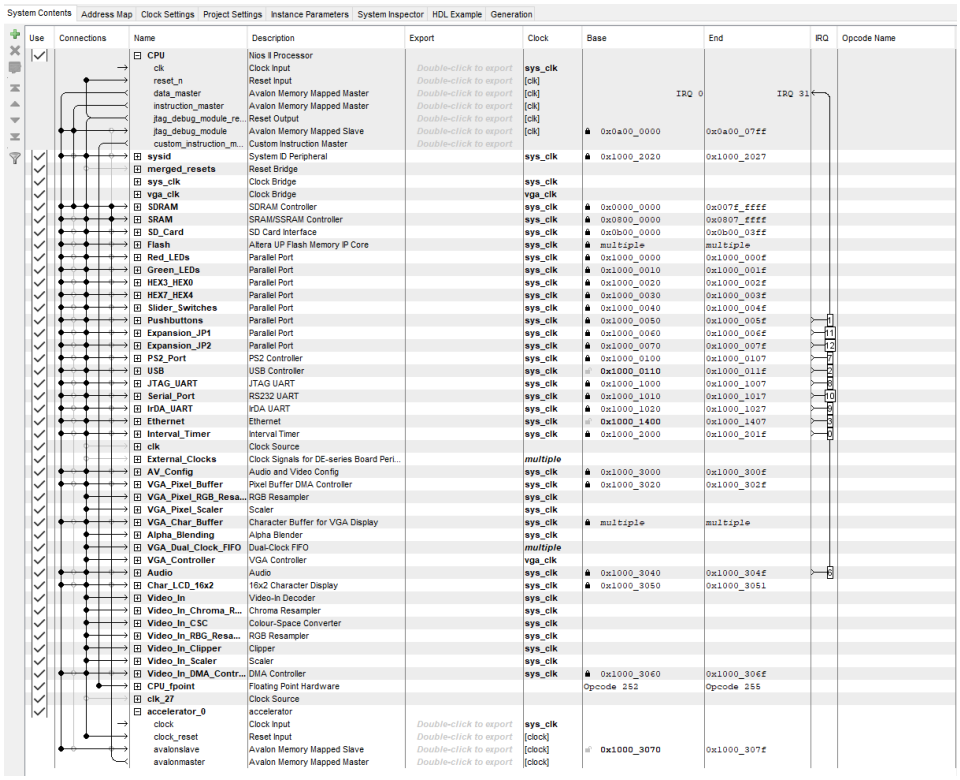
در تب Interfaces لازم است Address units مربوط به همه اینترفیس های موجود را به symbol تغییر دهیم، چون در کد وریلاگ رجیسترها 8 بیتی هستند و هر 32 بیت شامل 4 آدرس متوالی است. تنظیمات این بخش بایستی منطبق بر تصاویر زیر باشد:





البته در تصویر بالا برای pipelined transfers همان آدرس دهی پیش فرض را قرار دادم، که در این کد از burst استفاده نکردم، پس اهمیتی ندارد. پس از آن با کلیک بر روی remove interfaces with no signals و سپس finish اضافه کردن کامپوننت جدید را پایان میدهیم.

در ادامه با دبل کلیک بر روی کامپوننتی که اضافه کردیم، میتوانیم wiring مربوط به اینترفیس های کدمان را در محیط Qsys برقرار کنیم. بدین منظور بایستی کانفیگ مربوط به پورت های master و slave سخت افزاری که اضافه کردیم مطابق تصویر زیر باشد:



در پایان با مراجعه به تب Generation و کلیک بر روی Generate تغییرات سخت افزاری اعمال شده در محیط Qsys را به پروژه اضافه میکنیم.

پس از آن به پروژه در کوارتوس مراجعه کرده و مجددا کد پروژه را سنتز میکنیم تا تغییرات جدید اعمال شود. نتیجه سنتز آن به ترتیب زیر خواهد بود:

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Mon Jan 10 19:12:29 2022
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	20,859
Total combinational functions	16,630
Dedicated logic registers	12,809
Total registers	12809
Total pins	416
Total virtual pins	0
Total memory bits	170,297
Embedded Multiplier 9-bit elements	21
Total PLLs	2

در این مرحله تمام موارد مورد نیاز آماده است تا بتوانیم در کد C از سخت افزار اضافه شده استفاده کنیم. توابع مورد نیاز برای استفاده از این سخت افزار در system.h موجود هستند و با مراجعه به آن، اسمی که در محیط Qsys برای کامپوننت خود انتخاب کرده بودیم را سرچ میکنیم تا توابع مربوط به آنرا پیدا کنیم و از درست بودن فرآیند طی شده اطمینان حاصل کنیم. در صورت درست بودن فرآیند طی شده، در system.h، definition هایی مانند زیر را می‌یابیم:

```
/*
 * accelerator_0 configuration
 *
 */

#define ACCELERATOR_0_BASE 0x10003070
#define ACCELERATOR_0_IRQ -1
#define ACCELERATOR_0_IRQ_INTERRUPT_CONTROLLER_ID -1
#define ACCELERATOR_0_NAME "/dev/accelerator_0"
#define ACCELERATOR_0_SPAN 16
#define ACCELERATOR_0_TYPE "accelerator"
#define ALT_MODULE_CLASS_accelerator_0 accelerator
```

به سادگی میتوان دید، آدرس Base ای که برای شتابدهنده در نظر گرفته شده، با آدرسی که در Qsys به آن اختصاص داده شده بود یکسان است و تنها موردی که برای انجام این پروژه کافی است، دانستن آدرس شروع شتابدهنده در ساختار Memory map جهت دسترسی و تعامل با آن است. لذا با اضافه کردن

کتابخانه ی system.h به پروژه، بدنه ی کد amplitude_operation و سایر توابعی که در درون آن فراخوانی شده اند را تکمیل میکنیم. همچنین چون در کد از IORD و IOWR بهره بردیم، لازم است کتابخانه io.h هم به کد بی‌افزاییم.

بدنه ی تابع amplitude_operation به ترتیب زیر است:

```
volatile unsigned int* acc_base_addr = (unsigned int*)ACCELERATOR_0_BASE;
void amplitude_operation(unsigned int size, unsigned int num, volatile unsigned int* rbuff_addr, volatile unsigned int* lbuff_addr, volatile unsigned long long int* dest_addr)
{
    amplitude_circute_set_size(size);
    // also for your debugging make int amplitude_circute_get_size(); (optional)
    amplitude_circute_set_num(num);
    // also for your debugging make int amplitude_circute_get_num(); (optional)
    amplitude_circute_set_rbuff_addr(rbuff_addr);
    // also for your debugging make int amplitude_circute_get_lbuff_addr(); (optional)
    amplitude_circute_set_lbuff_addr(lbuff_addr);
    // also for your debugging make int amplitude_circute_get_rbuff_addr(); (optional)
    amplitude_circute_set_dest_addr(dest_addr);
    // also for your debugging make int amplitude_circute_get_dest_addr(); (optional)
    amplitude_circute_start();

    unsigned int reg0;

    //wait untill done bit sets to zero
    reg0=IORD(acc_base_addr, 0);
    while((reg0 & 0x80000000) == 0x80000000)
        reg0=IORD(acc_base_addr, 0);

    //setting start bit to zero
    reg0=IORD(acc_base_addr, 0);
    reg0=reg0 & 0xfffffffe;
    IOWR(acc_base_addr, 0, reg0);

    //wait untill done bit is issued
    reg0=IORD(acc_base_addr, 0);
    while((reg0 & 0x80000000) != 0x80000000)
        reg0=IORD(acc_base_addr, 0);

    return;
}
```

در این کد ابتدا سایز هر بخش بافر که میخواهیم بخوانیم در رجیستر 0 ثبت میشود، سپس تعداد بخش ها را در رجیستر 0 ثبت میکنیم. در گام بعد آدرس بافر های راست و چپ و در نهایت آدرسی که باید در آن حاصل را ذخیره کنیم را در رجیستر 3 ذخیره میکنیم. پس از انجام ستاپ رجیستر ها بیت start در رجیستر 0 را یک میکنیم تا محاسبات شروع شود. سپس منتظر میمانیم تا بیت done در رجیستر 0 برابر 0 شود(به معنای آنکه شتابدهنده متوجه شده است که باید محاسباتی را شروع کند). پس از آن بیت start را صفر میکنیم و منتظر میمانیم تا بیت done مجددا یک شود تا متوجه شویم محاسبات پایان یافته و نتیجه مورد نظر در آدرس مقصد ذخیره شده است. در ادامه در تصویر زیر هریک از توابعی که در بالا فراخوانی شده اند و توصیفشان را در این پاراگراف گفتیم را میتوانید مشاهده کنید:

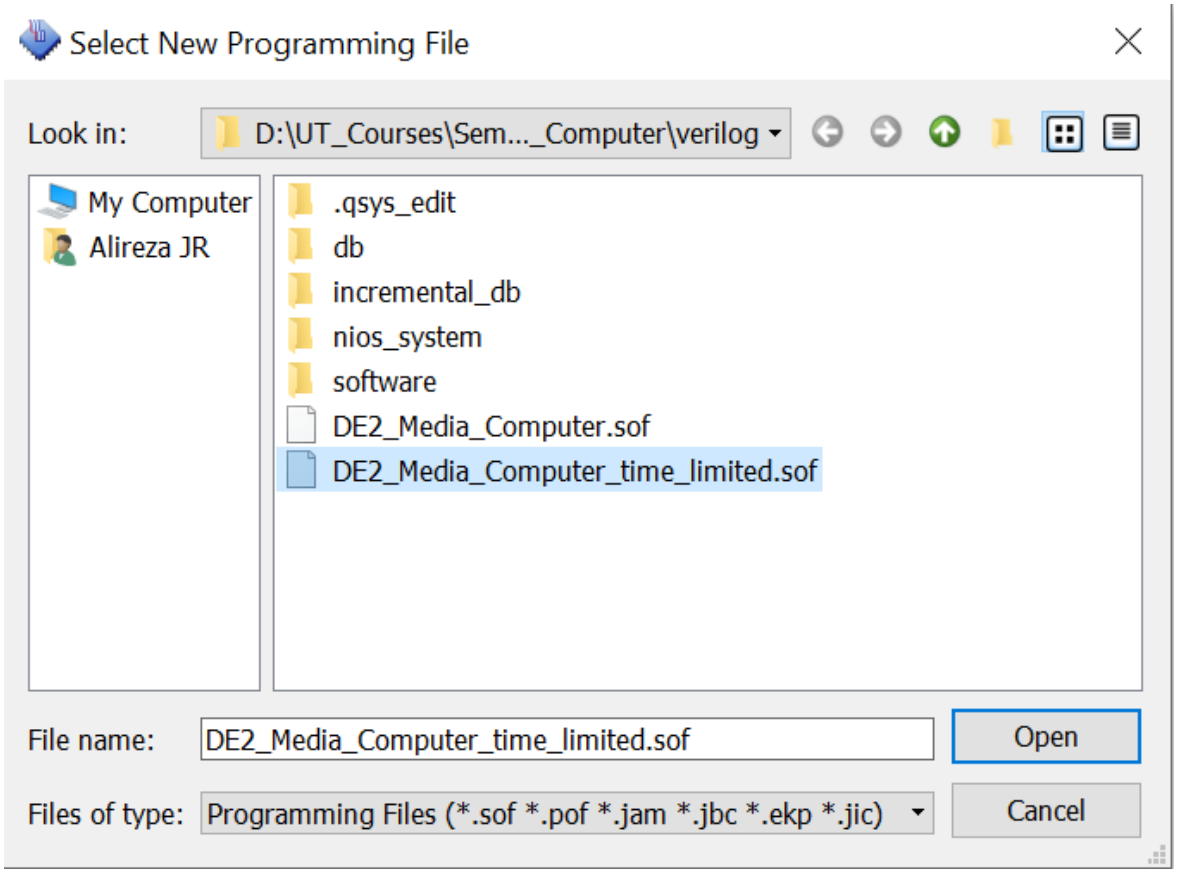
```
void amplitude_circute_set_size(unsigned int size){
    unsigned int reg0=0;
    reg0=reg0 | (size << 12);
    IOWR(acc_base_addr, 0, reg0);//rewrite new reg0:including size
    return;
}

void amplitude_circute_set_num(unsigned int num){
    unsigned int reg0;
    reg0=IORD(acc_base_addr, 0);
    reg0=reg0 & 0xfffff000; //clear num portion and start bit
    reg0=reg0 | (num << 1);
    IOWR(acc_base_addr, 0, reg0);//rewrite new reg0:including num,size
    return;
}

void amplitude_circute_set_rbuff_addr(volatile unsigned int* rbuff_addr){
    unsigned int reg1;
    reg1=(unsigned int) rbuff_addr;
    //reg1 = (unsigned int) &rbuff_addr[0];
    IOWR(acc_base_addr, 1, reg1);
    return;
}

void amplitude_circute_set_lbuff_addr(volatile unsigned int* lbuff_addr){
    unsigned int reg2;
    reg2=(unsigned int) lbuff_addr;
    IOWR(acc_base_addr, 2, reg2);
    return;
}

void amplitude_circute_set_dest_addr(volatile unsigned long long int* dest_addr){
    unsigned int reg3;
    reg3=(unsigned int) dest_addr;
    IOWR(acc_base_addr, 3, reg3);
    return;
}

void amplitude_circute_start(){
    unsigned int reg0;
    reg0=IORD(acc_base_addr, 0);
    reg0=reg0 | 0x00000001;
    IOWR(acc_base_addr, 0, reg0);//rewrite new reg0:including num,size,start
    return;
}
```

در ادامه از تابع amplitude_operation در بدنه لوپ موجود در main به ترتیب زیر بهره میگیریم:

```
//Hardware average calculations
alt_timestamp_start();
amplitude_operation((unsigned int)BUF_SIZE/N, N, r_buf, l_buf, hw_avg);
printf("Hardware Calculation Finished in %.3f seconds\n", (float)alt_timestamp()/(float)timestamp_freq);
```

hw_avg همان بافری است که حاصل محاسبات سخت افزاری در آن ذخیره میشود.

جهت ران کردن کد بر روی بورد، لازم است در بخش programmer ابتدا بر روی بورد ران شود تغییر کند به ، یعنی فایل هایلایت شده در تصویر زیر را باید انتخاب کنیم و روی بورد ران کنیم (توجه کنید پس از ران کردن یک وارنینگ نمایش داده میشود که تا زمانی که در حال اجرای کد خودتان بر روی بورد هستید، روی cancel کلیک نکنید!!!):



در ادامه کد را ران کردم و نتیجه ی ران شدن این بخش از کد به ترتیب زیر بود:

```
Hardware Calculation Finished in 0.301 seconds
```

اضافه کردن کد سخت افزاری، سرعت اجرای برنامه را حدود 11 برابر سریعتر کرد که در تصویر زیر میتوانید این دو زمان را در کنار یکدیگر مشاهده و با هم مقایسه کنید:

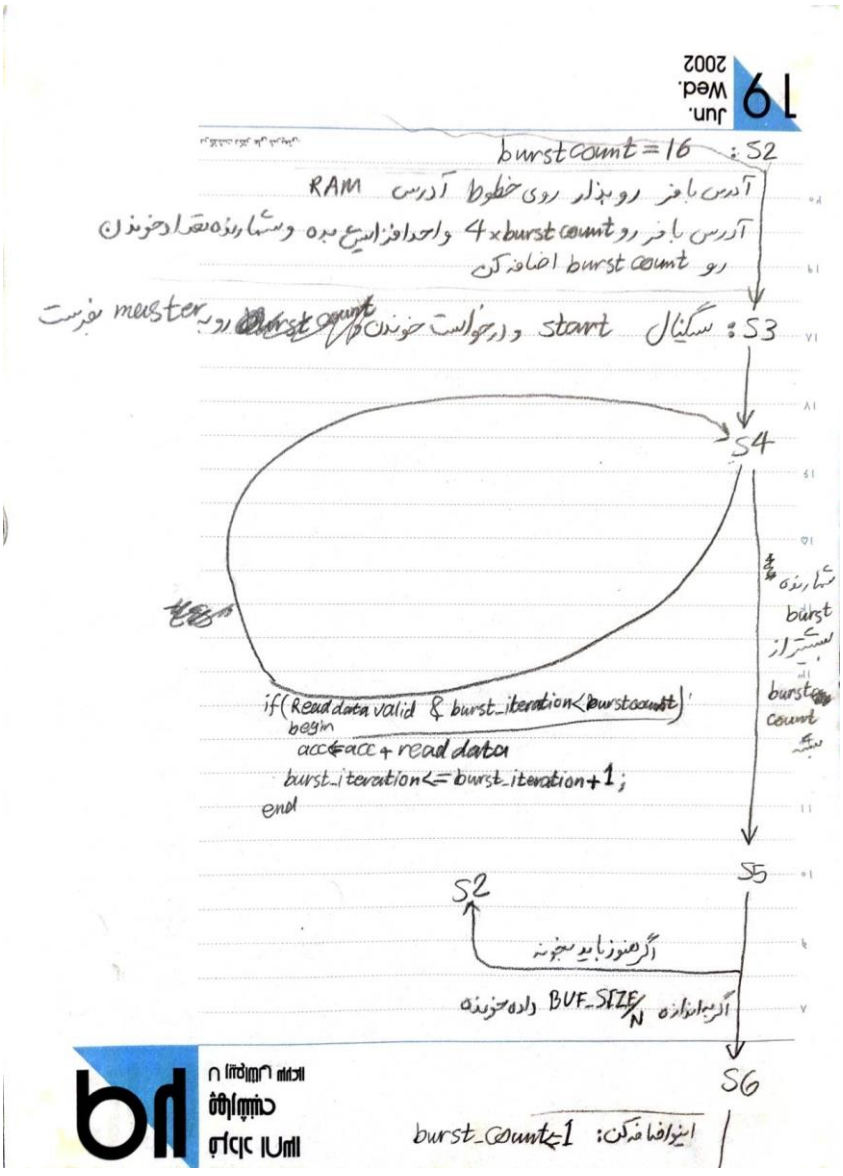
```
Calculation Started...
Software Calculation Finished in 3.378 seconds
Hardware Calculation Finished in 0.301 seconds
```

همچنین مقادیر حاصل انجام عمل جمع توسط کدهای نرم افزاری و سخت افزاری (که در آرایه های avg و hw_avg ذخیره شده بودند) را در تصویر زیر مشاهده میکنید که بسیار نزدیک به یکدیگر هستند و لذا عملکرد کد سخت افزاری درست است:

```
Sofware Calculation Result: 30180730
Hardware Calculation Result: 30170186
Sofware Calculation Result: 31819930
Hardware Calculation Result: 31823725
Sofware Calculation Result: 29578669
Hardware Calculation Result: 29505521
Sofware Calculation Result: 94884846
Hardware Calculation Result: 94893534
Sofware Calculation Result: 41295958
Hardware Calculation Result: 41368488
Sofware Calculation Result: 34649967
Hardware Calculation Result: 34595133
Sofware Calculation Result: 34240743
Hardware Calculation Result: 34236961
Sofware Calculation Result: 36046536
Hardware Calculation Result: 36111059
Sofware Calculation Result: 81319420
Hardware Calculation Result: 81256804
Sofware Calculation Result: 41888292
Hardware Calculation Result: 41930008
Sofware Calculation Result: 37222243
Hardware Calculation Result: 37234496
Sofware Calculation Result: 41846128
Hardware Calculation Result: 41813630
Sofware Calculation Result: 32378041
Hardware Calculation Result: 32382170
Sofware Calculation Result: 38598998
Hardware Calculation Result: 38501092
Sofware Calculation Result: 38134948
Hardware Calculation Result: 38201804
```

بخش امتیازی

در این بخش قصد داریم امکان Burst را به کد سخت افزاری خود بی‌افزاییم. بدین منظور کد های accelerator_master و کنترلر accelerator را لازم است تغییر دهیم. در استیت ماشینی که در گذشته طراحی کردیم، تغییراتی که در تصویر زیر میبینید را لازم است ایجاد کنیم(استیت های قبل و بعد از آن مانند گذشته باقی میمانند):



متاسفانه به دلیل کمبود وقت، نتوانستم وقت کافی برای این بخش اختصاص دهم و بخشی از کد کنترلر را منطبق بر تغییرات بالا، تغییر دادم و بیش از طراحی کنترلر جدید و انجام یکسری تغییرات در کد های وریلاگ نتوانستم پیشروی بیشتری داشته باشم. اما در نهایت انتظار میرود اجرای برنامه توسط burst به طرز چشم گیری به لحاظ سرعت اجرا بهبود یابد، چون دیگر لازم نیست به ازای خواندن هر داده، شتابدهنده

با SDRAM سوال و جواب کند و در این حالت، در هر گام به جای 1 داده، 16 داده دریافت شده و با هم جمع میشود.

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 hardware به سیستم خود اضافه کنیم. با اضافه شدن دستورات و definition های اختصاصی آن سخت افزار در کد C برای انجام عملیات مورد نظر، آن عملیات را به صورت سخت افزاری با کانفیگ سخت افزاری مخصوص اجرای آن دستور، بر اساس آنکه مسئله ی ما چه باشد، میتواند با سرعت بسیار بالاتری در مقایسه با روش نرم افزاری انجام شود.

در این آزمایش هم مشاهده کردیم که اجرای نرم افزاری عملیات جمع کردن مقادیر بافر حدود 3.3 ثانیه به طول انجامید در حالی که اجرای آن با استفاده از دستور اختصاصی، در حدود 0.3 ثانیه به اتمام رسید که افزایش سرعت چشمگیر محاسبات سخت افزاری در مقایسه با محاسبات نرم افزاری در ایجاد دسترسی به RAM و انجام عمل جمع را نشان میدهد.