

بخش اول: MicroBlaze & SDK

1. MicroBlaze

- توضیحات مربوط به بلوک‌های مختلف IP Core MicroBlaze:

MicroBlaze یک پردازنده قابل تنظیم است که می‌تواند برای رفع نیازهای خاص یک برنامه طراحی شود و به همین علت شامل طیف گسترده‌ای از ویژگی‌ها، از جمله یک واحد مدیریت حافظه (MMU)، ضرب‌کننده و تقسیم‌کننده سخت‌افزاری و انواع رابط برای اتصال به اجزای دیگر در یک FPGA است. MicroBlaze همچنین از انواع برنامه‌های افزودنی مجموعه دستورالعمل‌ها، از جمله دستورالعمل‌های ممیز شناور و بردار پشتیبانی می‌کند که در صورت نیاز می‌توان آنها را اضافه کرد. بنابراین، برخی از بلوک‌ها جزئی از اجزای ثابت MicroBlaze هستند و برخی دیگر را می‌توان به صورت انتخابی به آن اضافه کرد. برخی از این بلوک‌ها عبارتند از:

○ هسته پردازنده (Processor core)

- i. واحد پردازشی اصلی MicroBlaze است که عملیات محاسباتی و منطقی را روی داده‌ها انجام می‌دهد. این واحد شامل ویژگی‌های مختلفی است که در ادامه به صورت خلاصه معرفی می‌شوند:
- ii. معماری مجموعه دستورالعمل (ISA): هسته پردازنده MicroBlaze از یک ISA مبتنی بر RISC استفاده می‌کند که برای عملکرد بالا و مصرف انرژی کم بهینه شده است.
- iii. اجرای Pipeline: هسته پردازنده MicroBlaze از یک معماری اجرای Pipeline استفاده می‌کند که اجازه می‌دهد چندین دستورالعمل به طور همزمان پردازش شوند و عملکرد را بهبود بخشد.
- iv. واحد مدیریت حافظه (MMU): هسته پردازنده MicroBlaze شامل یک MMU است که از حافظه مجازی پشتیبانی می‌کند و به پردازنده اجازه می‌دهد به حافظه فراتر از فضای آدرس فیزیکی دسترسی پیدا کند.
- v. وقفه‌ها: هسته پردازنده MicroBlaze از وقفه‌ها پشتیبانی می‌کند و به آن اجازه می‌دهد به رویدادهای خارجی مانند سرریز تایمر یا رویدادهای ورودی/خروجی پاسخ دهد.
- vi. ضرب و تقسیم سخت‌افزار: هسته پردازنده MicroBlaze شامل پشتیبانی سخت‌افزاری برای عملیات ضرب و تقسیم است که عملکرد برنامه‌هایی را که به این عملیات نیاز دارند بهبود می‌بخشد.
- vii. برنامه‌های افزودنی مجموعه دستورالعمل‌های قابل تنظیم: هسته پردازنده MicroBlaze از طیف وسیعی از برنامه‌های افزودنی مجموعه دستورالعمل، از جمله

دستورالعمل‌های ممیز شناور و بردار پشتیبانی می‌کند، که می‌توانند در صورت نیاز اضافه شوند.

○ Instruction Cache

یک حافظه نهان است که دستورالعمل‌های پرکاربرد را برای بهبود عملکرد پردازنده ذخیره می‌کند.

○ Data Cache

یک حافظه نهان است که داده‌های پرکاربرد را برای بهبود عملکرد پردازنده ذخیره می‌کند.

○ واحد کنترل وقفه (Interrupt Controller)

این واحد تمام وقفه‌های تولید شده توسط سیستم را مدیریت کرده و آنها را به کنترل‌کننده مناسب هدایت می‌کند.

○ Timer

این واحد عملکرد تایمر را برای پردازنده فراهم می‌کند و به آن امکان می‌دهد تا زمان را برای عملیات‌های مبتنی بر زمانبندی محاسبه کند.

○ واحد Debugging

این واحد پشتیبانی از Debugging را فراهم می‌کند و به توسعه‌دهندگان این امکان را می‌دهد تا بخشی از کدهای خود را که در MicroBlaze اجرا می‌شود اشکال‌زدایی کنند.

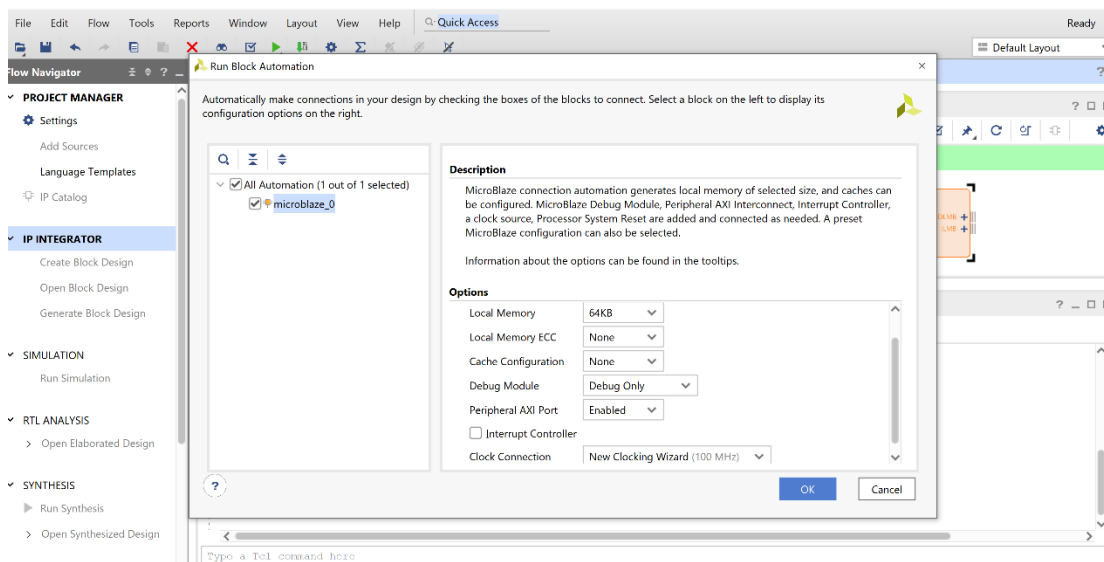
○ Bus Interface Unit (BIU)

این واحد رابط بین پردازنده و حافظه خارجی و تجهیزات جانبی را فراهم می‌کند.

همچنین بلوک‌های دیگری مانند واحد ممیز شناور (FPU)، واحد دسترسی مستقیم به حافظه (DMA)، واحد کنترل‌کننده UART و ... وجود دارند که برای استفاده از آنها باید زمان افزودن MicroBlaze IP Core تنظیمات مربوط به افزودن آنها را انجام داد.

• تنظیم بلوک‌ها در Vivado

برای این کار، هنگام اضافه کردن Block Design جدید از نوع MicroBlaze به پروژه، با پنجره‌ای مواجه می‌شویم که به کمک آن می‌توانیم تنظیمات اولیه مانند تعیین مقدار حافظه محلی، تنظیمات مربوط به کش، ماژول Debugger، فرکانس کلاک و ... را انجام دهیم:

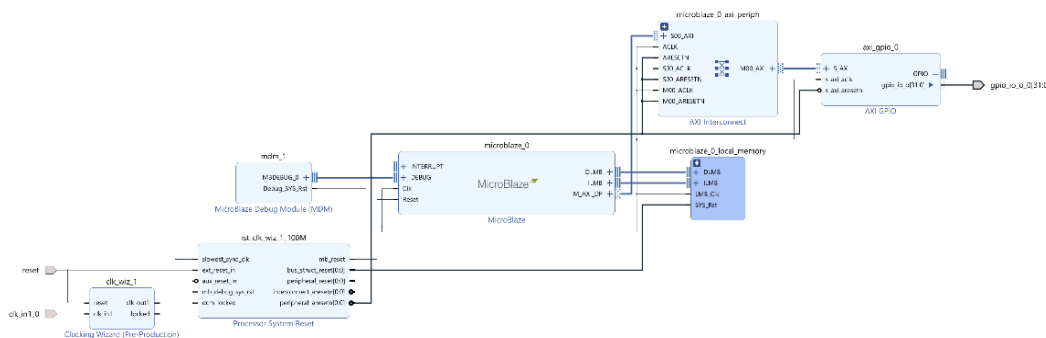


بعد از اعمال این تغییرات، سایر بلوک‌ها به صورت خودکار به بلوک اصلی پردازنده MicroBlaze متصل می‌شوند و در ادامه با کلیک کردن بر روی هر یک، تنظیمات مخصوص به آن نمایش داده می‌شود. نکته حائز اهمیت آن است که تنظیمات هر بلوک می‌تواند از سایر بلوک‌ها کاملاً متفاوت باشد و دستور کار یکسانی برای Configuration تمام بلوک‌ها وجود ندارد. همچنین در این مرحله می‌توانیم سایر بلوک‌های انتخابی که توسط MicroBlaze پشتیبانی می‌شوند مانند FPU، GPIO، Ethernet و... را اضافه کرده و اتصالات آنها با میکرو به صورت دستی یا اتوماتیک برقرار نماییم.

2. روشن و خاموش کردن GPIO

- آماده‌سازی پروژه و ساخت بلوک دی‌گرام:

پس از افزودن یک بلوک MicroBlaze به پروژه و انجام تنظیمات مربوط به کلاک ورودی، یک بلوک IP Core جدید به نام GPIO اضافه می‌کنیم. در تنظیمات این بلوک، سائز GPIO را جهت تست روی 32 تنظیم کرده و ضمناً دقت می‌کنیم تا باس مربوطه از نوع خروجی باشد. سپس یک پورت جدید برای خروجی GPIO تعریف کرده و نهایتاً سایر اتصالات میان بلوک‌ها را تکمیل می‌نماییم.



- توضیحات کد نوشته‌شده در SDK به زبان C:

در ابتدا باید کتابخانه‌هایی که Xilinx در اختیار ما قرار داده و به کمک آنها می‌توان دستورات قابل اجرا در میکروبلیز را فراخوانی کرد، در کد اصلی include کنیم. همچنین برای خوانایی بیشتر کد، ID بخش‌های خارجی با عنوان OUTPUT_DEVICE_ID تعریف شده است:

```
#include "xparameters.h"
#include "xgpio.h"
#define OUTPUT_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
```

قبل از شروع اجرای برنامه باید دستورات مربوط به initialize کردن باس GPIO قرار گیرند.
به این منظور دستورات زیر قبل از حلقه while اصلی نوشته شده‌اند:

```
//Set output Status
int outputStatus;
outputStatus = XGpio_Initialize (&outputGpio, OUTPUT_DEVICE_ID);

//Set the direction of the GPIO channels
//Set output channel as output (32-bit width)
XGpio_SetDataDirection (&outputGpio, 1, 0x00000000);

//Set Initial Value of GPIO to 0X00000001
XGpio_DiscreteWrite (&outputGpio, 1, 1);
```

قطعه اصلی برنامه که باید در طول اجرا تکرار شود، درون حلقه While قرار می‌گیرد.

- ساخت Testbench و بررسی نتایج شبیه‌سازی:

برای تست خروجی GPIO، درون حلقه بر روی پایه‌های GPIO (که مقدار اولیه آن را X00000001 در نظر گرفته بودیم) مقدار X00000000 را می‌نویسیم و سپس با دستور MB_Sleep به اندازه 1 میلی‌ثانیه تاخیر ایجاد کرده و مجدداً خروجی GPIO را به 1 تغییر می‌دهیم.

```
while (1)
{
    XGpio_DiscreteWrite(&outputGpio, 1, 0);
    MB_Sleep(1);

    XGpio_DiscreteWrite(&outputGpio, 1, 1);
    MB_Sleep(1);
}
```

به این ترتیب، در فواصل یک میلی‌ثانیه‌ای خروجی GPIO از 0 به 1 تغییر مقدار می‌دهد. برای تست کردن این قطعه کد، فایل دارای پسوند elf که توسط نرم‌افزار SDK ساخته شده‌است را به پروژه اصلی اضافه می‌کنیم.

پس از آن یک testbench ساخته و بعد از اضافه کردن component های مورد نیاز، تعریف سیگنال‌ها و پریود کلاک و انجام port map، ابتدا پروسه مربوط به تولید کلاک ورودی را مطابق زیر پیاده‌سازی می‌کنیم:

```

clk_process : process
begin
    clk_100MHz <= '0';

    wait for clk_period/2;
    clk_100MHz <= '1';

    wait for clk_period/2;
end process clk_process;

```

نکته: با توجه به اینکه در بلوک مربوط به کلاک، فرکانس پیشفرض (100 MHz) انتخاب شد، در تست‌بنچ ساخته شده تناوب کلاک را 10 ns در نظر گرفتیم.

در ادامه، ابتدا برای 100 ns ریست را 1 نگه می‌داریم و پس از آن ریست را 0 می‌کنیم. بنابراین انتظار داریم تا پس از گذشت 1 ms از زمان 0 شدن سیگنال ریست، سیگنال خروجی GPIO مقدار 32 بیتی 1 را اتخاذ کند و پس از آن به تناوب toggle شود.

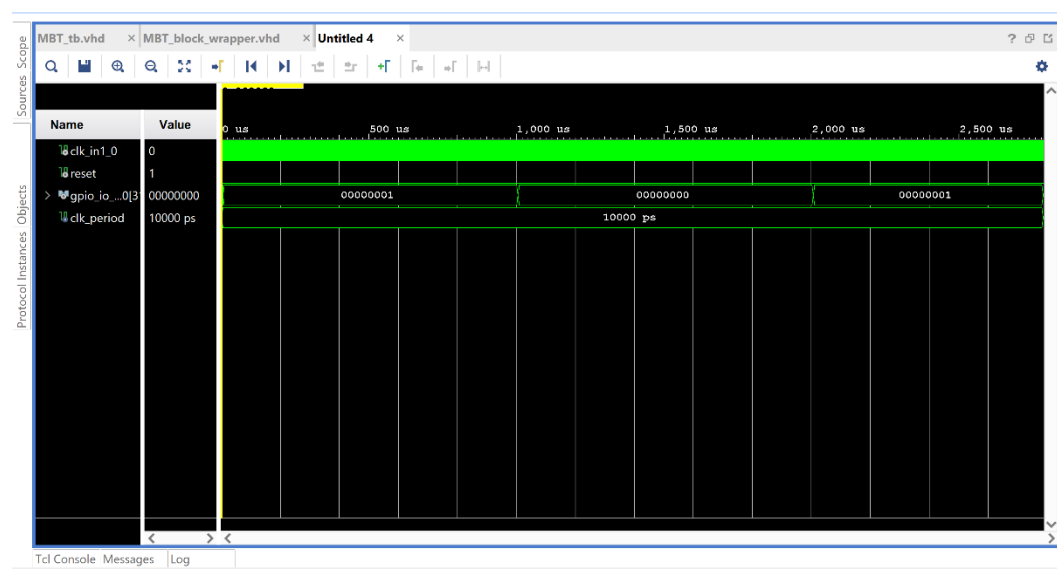
کد مربوط به تست GPIO و خروجی آن در ادامه آورده شده است:

```

stimulus_process : process
begin
    reset <= '1';
    wait for 100 ns
    reset <= '0';
    wait;
end process stimulus_process;

```

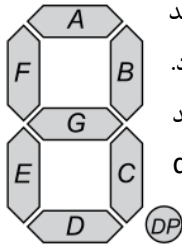
خروجی شبیه‌سازی:



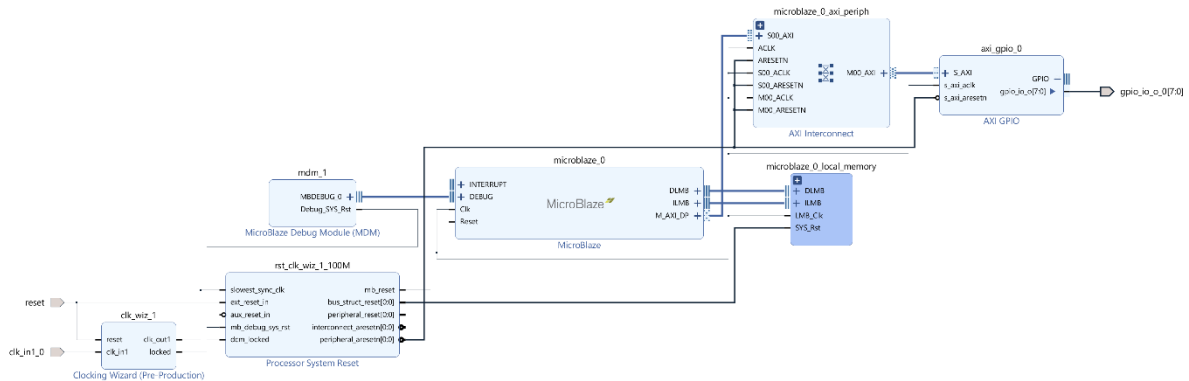
ملاحظه می‌شود که مطابق انتظار، سیگنال خروجی GPIO در فواصل 1ms تغییر مقدار داده است.

3. کنترل 7-SEGMENT تک رقمی

- آماده سازی پروژه و ساخت بلوک دیاگرام:



تمامی مراحل ساخت پروژه و طراحی بلوک دیاگرام برای این قسمت از پروژه، همانند بخش قبل انجام می شود، با این تفاوت که اندازه GPIO برابر با 8 در نظر گرفته می شود. دلیل این کار آن است که خروجی GPIO در واقع ورودی ماژول 7-Segment خواهد بود. این ماژول 7 ورودی برای ال ای دی های a تا g و یک ورودی برای ال ای دی dp (نقطه) دریافت می کند، که در این تست همواره خاموش خواهد بود.



- توضیحات کد نوشته شده در SDK به زبان C:

مراحل initialize کردن GPIO مانند بخش قبل که توضیح داده شد، انجام می پذیرد. در ادامه و در حلقه while اصلی، کد مربوط به شمارش و تولید سیگنال خروجی نوشته شده است. در یک حلقه for، یک شمارشگر را از 0 تا 9 افزایش می دهیم و هر بار با توجه به رقمی که در حال شمارش است، کد 8 بیتی که باید به 7-Seg ارسال شود را به کمک Switch case تولید می کنیم. ضمناً هر بار بعد از تعیین 8 بیت خروجی، 1 ms صبر می کنیم تا تغییر خروجی در شبیه سازی قابل رویت باشد:

```
while (1)
{
    for (int i = 0; i < 10; i ++)
    {
        switch (i)
        {
            case 0:
                XGpio_DiscreteWrite(&outputGpio,1, 0b11111100);
                MB_Sleep(1);
                break;

            case 1:
                XGpio_DiscreteWrite(&outputGpio,1, 0b10110000);
                MB_Sleep(1);
                break;

            case 2:
                XGpio_DiscreteWrite(&outputGpio,1, 0b11011010);
                MB_Sleep(1);
```

```

        break;

        case 3:
            XGpio_DiscreteWrite(&outputGpio,1, 0b11110010);
            MB_Sleep(1);
            break;

        case 4:
            XGpio_DiscreteWrite(&outputGpio,1, 0b01100110);
            MB_Sleep(1);
            break;

        case 5:
            XGpio_DiscreteWrite(&outputGpio,1, 0b10110110);
            MB_Sleep(1);
            break;

        case 6:
            XGpio_DiscreteWrite(&outputGpio,1, 0b10111110);
            MB_Sleep(1);
            break;

        case 7:
            XGpio_DiscreteWrite(&outputGpio,1, 0b11100000);
            MB_Sleep(1);
            break;

        case 8:
            XGpio_DiscreteWrite(&outputGpio,1, 0b11111110);
            MB_Sleep(1);
            break;

        case 9:
            XGpio_DiscreteWrite(&outputGpio,1, 0b11110110);
            MB_Sleep(1);
            break;
    }

}

}

```

- ساخت Testbench و بررسی نتایج شبیه‌سازی:
 برای شبیه‌سازی نتایج این بخش، بعد از تعریف component ها و سیگنالها و انجام port map، تست‌بنچی دقیقا مطابق با بخش قبل می‌نویسیم. این بار پیش‌بینی می‌کنیم بعد از 0 شدن سیگنال ریست، کدهای 8 رقمی مربوط به ارقام 0 تا 9 را در فواصل 1 ms ببینیم:

- توضیح کد زبان C:

با توجه به توضیحاتی که پیش تر داده شد، در این قسمت باید از 0 تا 99 را با فرکانس مشخصی (مثلا نیم ثانیه) شمارش کنیم و هر 2ms یکبار خروجی مربوط به سون سگمنت یکان یا دهگان را به همراه دو بیت کنترلی تعیین کرده و به ورودی سون سگمنت بفرستیم.

به این منظور، بعد از انجام مراحل اولیه که در بخشهای قبل شرح داده شدند، ابتدا متغیرهای مربوط به رقم یکان، رقم دهگان، تعیین سون سگمنت (سیگنال کنترلی) و شمارش مجموع تاخیر ایجادشده در اجرای برنامه را تعریف می کنیم. (بخش بنفش رنگ)

سپس در حلقه while اصلی، ابتدا در صورتی که رقم یکان به 10 رسیده باشد آن را صفر کرده و دهگان را یکی افزایش می دهیم. (رنگ آبی)

سپس بررسی می کنیم که سیگنال خروجی در این لحظه از اجرای کد برای سون سگمنت یکان ساخته می شود یا دهگان؛ و در هر صورت تابع ساخت این خروجی را فراخوانی کرده و سپس مقدار متغیر کنترلی را بین 0 و 1 toggle کرده و 2ms تاخیر ایجاد می کند. به این ترتیب پس از اجرای مجدد حلقه، خروجی برای سون سگمنت دیگر تولید خواهد شد. (قسمت سبز)

با فرض اینکه تناوب شمارش را هر 0.5 ثانیه یکبار در نظر گرفته ایم (برای اینکه اعداد مختلف با چشم قابل تشخیص باشند) بررسی می کنیم که آیا مجموعا 250 بار تابع وقفه 2ms را اجرا کرده ایم یا خیر؛ در صورتی که این شرط برقرار باشد مجاز هستیم شمارش را یکی افزایش دهیم و در غیر این صورت همچنان باید عدد دو رقمی قبل روی سون سگمنت ها نمایش داده شود (قسمت زرد)

```
int counter_ones = 0;
int counter_tens = 0;
int control = 0;
int delay_counter = 0;
while(1)
{
    if (counter_ones == 10)
    {
        counter_ones = 0;
        counter_tens++;
    }

    if (control == 0)
    {
        seven_seg_update(counter_ones, control);
        control = 1;
    }
    else
    {
        seven_seg_update(counter_tens, control);
        control = 0;
    }
    MB_Sleep(2);
    delay_counter++;

    if (delay_counter > 250)
```

```

{
    counter_ones ++;
    delay_counter = 0;
}
}

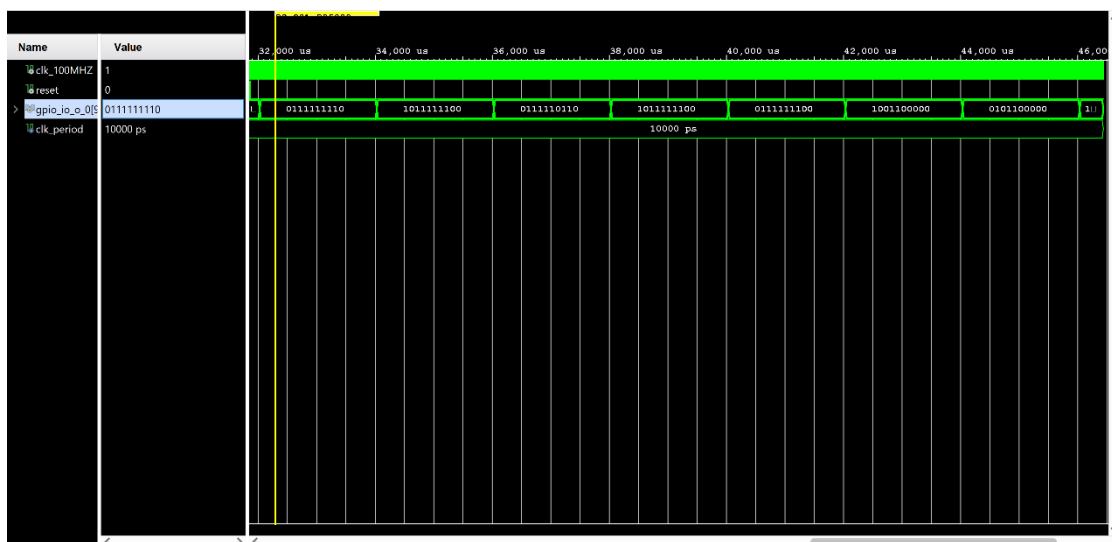
```

همچنین تابع مربوط به سون‌سگمنت پایین تابع main نوشته شده است. در این تابع بسته به اینکه رقم در حال نمایش یکان یا دهگان است (که توسط متغیر control تعیین می‌شود) دو بیت سمت راست مقدار 01 یا 10 را اتخاذ می‌کنند. سپس 8 بیت بعد با کد مربوط به نمایش ارقام 0 تا 9 تعیین می‌شوند. کد مربوط به این تابع به دلیل طولانی بودن، در گزارش کار آورده نشده و برای بررسی آن می‌توان به بخش آخر از فایل sevenSeg2_c مراجعه کرد.

- ساخت Testbench و بررسی نتایج شبیه‌سازی:

برای پیاده‌سازی ساده‌تر شبیه‌سازی، فرض بر این قرار گرفت که شمارش اعداد 0 تا 99 با تناوب 2 ms انجام می‌پذیرد (بنابراین عدد 250 در کد قبل برای سادگی با 1 جایگزین شد). سپس فایل تست‌بنچ ساخته شده و مانند بخش‌های پیشین بعد از 100ns مقدار ریست برابر با 0 قرار داده شد. برای مشاهده بهتر نتایج لازم است تا شبیه‌سازی را برای مدت بیشتری اجرا کنیم تا تغییرات سیگنال‌ها به خوبی دیده شود. در این شبیه‌سازی توقع داریم عدد در حال نمایش در فواصل 10ms تغییر کند و ضمناً در فواصل 2ms دو بیت سمت راست خروجی از 01 به 10 و برعکس تغییر مقدار دهد (که به معنای روشن شدن شدن متناوب سون‌سگمنت هاست).

بعد از انجام شبیه‌سازی، نتایج مطلوب حاصل شد (تصویر زیر مربوط به نمایش 08 تا یکان عدد 11 است):



بخش دوم: Vivado HLS

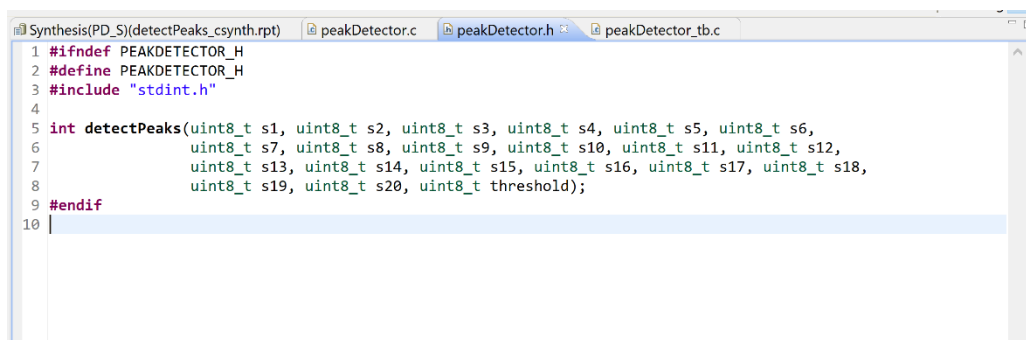
1. طراحی قله‌شمار:

در نرم‌افزار Vivado HLS، برای هر IP-Core مجموعه‌ای فایل نیاز خواهیم داشت که کد مربوط به هریک از این فایل‌ها برای Peak Detection IP Core در ادامه به تفکیک بررسی می‌شوند:

- فایل Header (peakDetector.h)

در این فایل declaration مربوط به توابع C و تعاریف سطح macro که بین منابع مختلف به صورت مشترک مورد استفاده قرار می‌گیرند نوشته می‌شود.

بنابراین فراخوانی و ورودی/خروجی‌های تابع detectPeaks در این فایل قرار دارند. بعداً با فراخوانی هدر peakDetector.h در فایل کد زبان C، می‌توانیم از این تابع استفاده کنیم.



```
1 #ifndef PEAKDETECTOR_H
2 #define PEAKDETECTOR_H
3 #include "stdint.h"
4
5 int detectPeaks(uint8_t s1, uint8_t s2, uint8_t s3, uint8_t s4, uint8_t s5, uint8_t s6,
6               uint8_t s7, uint8_t s8, uint8_t s9, uint8_t s10, uint8_t s11, uint8_t s12,
7               uint8_t s13, uint8_t s14, uint8_t s15, uint8_t s16, uint8_t s17, uint8_t s18,
8               uint8_t s19, uint8_t s20, uint8_t threshold);
9 #endif
10
```

- فایل کد اصلی (peakDetector.c)

در این فایل بخش اصلی کد که شامل تعریف تابع detectPeaks است قرار می‌گیرد. در ورودی، سیگنال را که شامل 20 مقدار است دریافت می‌کنیم، و این مقادیر را درون آرایه‌ای به نام signal ذخیره می‌کنیم. در ادامه، یک متغیر 32 بیتی تعریف می‌کنیم که از 4 بیت اول آن برای ذخیره تعداد پیک‌ها و از سایر بیت‌های آن برای تعیین محل وقوع پیک استفاده خواهیم کرد. به این ترتیب تمام درایه‌های آرایه signal را توسط یک حلقه for چک می‌کنیم و در صورتی که مقدار آن از دو درایه قبل و بعد از خود بیشتر بوده و ضمناً از مقدار آستانه تعریف شده نیز بیشتر باشد، آن را به عنوان پیک تشخیص می‌دهیم. بعد از تشخیص پیک، به متغیر peaks عدد 1 را اضافه می‌کنیم (که این تغییر در 4 بیت اول اعمال خواهد شد) که به معنای یک افزایش در تعداد پیک‌های شناسایی شده است. همچنین برای تشخیص محل وقوع پیک در یکی از 20 بیت بعدی، peaks را با عدد 1 که به اندازه 4+ (اندیس پیک) شیفت منطقی داده شده است جمع می‌کنیم، چرا که قبلاً توضیح دادیم که چهار بیت نخست متعلق به تعداد پیک‌ها و نه محل آنها هستند. (ضمناً بجای استفاده از عملکرد شیفت، از توان‌های عدد 2 استفاده کرده‌ایم که نمایش باینری یکسانی با 1 شیفت‌یافته دارند).

```

1 #include "peakDetector.h"
2 #include <math.h>
3 #include "stdint.h"
4
5 int detectPeaks(uint8_t s1, uint8_t s2, uint8_t s3, uint8_t s4, uint8_t s5, uint8_t s6,
6               uint8_t s7, uint8_t s8, uint8_t s9, uint8_t s10, uint8_t s11, uint8_t s12,
7               uint8_t s13, uint8_t s14, uint8_t s15, uint8_t s16, uint8_t s17, uint8_t s18,
8               uint8_t s19, uint8_t s20, uint8_t threshold)
9 {
10     uint8_t signal[20] = {s1, s2, s3, s4, s5, s6,
11                          s7, s8, s9, s10, s11, s12,
12                          s13, s14, s15, s16, s17, s18, s19, s20};
13     int peaks = 0x00000000;
14
15     for(int i = 1; i < 19; i++)
16     {
17         if(signal[i] > threshold && signal[i] > signal[i-1] && signal[i] > signal[i+1])
18         {
19             peaks++;
20             peaks = peaks + pow(2, (4+i));
21         }
22     }
23
24     return peaks;
25 }
26

```

• فایل تست (peakDetector_tb.c)

جهت تست عملکرد تابعی که در بخش قبل شرح داده شد، یک فایل تست بنویسیم و در آن متغیرهای s1 تا s20 را با مقادیر آزمایشی مقداردهی می‌کنیم. همچنین اندازه آستانه برای وقوع پیک را (Threshold) برابر نصف حداکثر اندازه سیگنال یعنی 127 در نظر می‌گیریم. در ادامه این مقادیر را به عنوان ورودی به تابع می‌دهیم و خروجی آن را به کمک دستور printf نمایش می‌دهیم. با بررسی این خروجی می‌توانیم از صحت عملکرد تابع نوشته شده مطمئن شویم.

```

1 #include <stdio.h>
2 #include "peakDetector.h"
3 #include "stdint.h"
4
5 int main()
6 {
7     uint8_t s1=52, s2=27, s3=167, s4=252, s5=8, s6=74,
8           s7=1, s8=24, s9=0, s10=19, s11=18, s12=30,
9           s13=17, s14=32, s15=19, s16=9, s17=85, s18=255, s19=18, s20=18;
10    uint8_t threshold = 127;
11
12    int peaks = detectPeaks(s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12,
13                          s13, s14, s15, s16, s17, s18, s19, s20, threshold);
14
15    printf("Peaks' report: %d\n", peaks);
16
17    return 0;
18 }
19

```

```

INFO: [APCC 202-3] Tmp directory is C:/tmp/apcc_db_USER/39162320605736
INFO: [APCC 202-1] APCC is done.
Generating csim.exe
Number of peaks: 2
Peaks occur at: 3 17 INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
Finished C simulation.

```

2. طراحی سنجش ضربان قلب

- فایل (calcHeartRate.h) Header

مطابق توضیحاتی که داده شد، declaration تابع calcHeartRate را انجام می‌دهیم.

```
Synthesis(CHR_Solu)(calcHeartRate_csynth.rpt)  calcHeartRate.c  calcHeartRate.h  calcHeartRate_tb.c
1 #ifndef CALCHEARTRATE_H
2 #define CALCHEARTRATE_H
3
4 int calcHeartRate(int peaks, int fs);
5
6 #endif
7
```

- فایل کد اصلی (calcHeartRate.c)

در بخش قبل دیدیم که اطلاعات مربوط به پیک‌های سیگنال در متغیر peaks ذخیره می‌شود. بنابراین این متغیر را به عنوان ورودی به تابع محاسبه ضربان قلب می‌دهیم. دیگر ورودی این تابع، فرکانس نمونه‌برداری خواهد بود.

در ابتدا تعداد پیک‌ها را به کمک ماسک کردن 4 بیت اول peaks مشخص کرده و در متغیر num می‌ریزیم. در ادامه یک آرایه به نام rPeaks برای نگهداشتن دو پیکی که فاصله بین آنها را محاسبه خواهیم کرد تعریف می‌کنیم و درون یک حلقه for، در صورتی که به اندیس یکی از پیک‌ها برسیم، مقدار این اندیس را درون آرایه می‌ریزیم.

در ادامه، یک حلقه for دیگر را به اندازه تعداد کل پیک‌ها تکرار می‌کنیم و در هر تکرار، فاصله هر پیک را با پیک قبلی حساب کرده و مجموع این فواصل را در متغیر rSum ذخیره می‌کنیم.

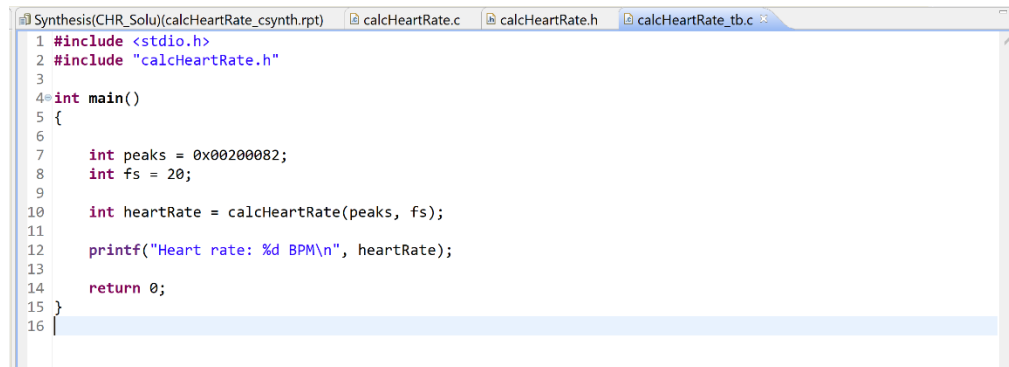
در نهایت با انجام محاسبات، خروجی نهایی را در متغیر heartRate ذخیره کرده و به عنوان خروجی تابع آن را برمی‌گردانیم.

```
Synthesis(CHR_Solu)(calcHeartRate_csynth.rpt)  calcHeartRate.c  calcHeartRate.h  calcHeartRate_tb.c
1 #include "calcHeartRate.h"
2 #include <math.h>
3
4 int calcHeartRate(int peaks, int fs) {
5     int rSum = 0;
6     int i, k;
7     int j = 0;
8     int rPeaks[2];
9     int num = peaks & 0x0000000F;
10
11     for(i = 1; i < 19; i++)
12     {
13         if(((int)peaks & (int)pow(2, (4 + i))) != 0)
14         {
15             rPeaks[j] = i;
16             j++;
17         }
18     }
19
20     for (k = 1; k < num; k++) {
21         rSum += rPeaks[k] - rPeaks[k - 1];
22     }
23
24     float avgRRInterval = (float)rSum / (num - 1);
25     int heartRate = fs / avgRRInterval * 60;
26
27     return heartRate;
28 }
29
```

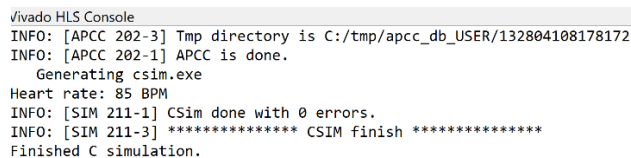
- فایل تست (calcHeartRate_tb.c)

برای تست کردن تابع calcHeartRate، فرض می‌کنیم در سیگنال ورودی دو پیک داشته‌ایم که در مکان 4 و 13 اتفاق افتاده‌اند. بنابراین بیت چهارم، دوازدهم و هفدهم متغیر peaks را برابر یک قرار می‌دهیم. ضمناً فرکانس نمونه‌برداری را برابر با 20 در نظر می‌گیریم. این دو متغیر را به عنوان ورودی به تابع داده و خروجی آن را (که انتظار داریم عدد 85 باشد) چاپ می‌کنیم.

در ادامه کد تست‌بنچ و همچنین مقدار چاپ شده در کنسول نرم‌افزار که برابر با مقدار صحیح است، آورده شده‌است:



```
1 #include <stdio.h>
2 #include "calcHeartRate.h"
3
4 int main()
5 {
6
7     int peaks = 0x00200082;
8     int fs = 20;
9
10    int heartRate = calcHeartRate(peaks, fs);
11
12    printf("Heart rate: %d BPM\n", heartRate);
13
14    return 0;
15 }
16
```



```
Vivado HLS Console
INFO: [APCC 202-3] Tmp directory is C:/tmp/apcc_db_USER/132804108178172
INFO: [APCC 202-1] APCC is done.
Generating csim.exe
Heart rate: 85 BPM
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
Finished C simulation.
```

بخش سوم: جمع آوری بخش‌ها

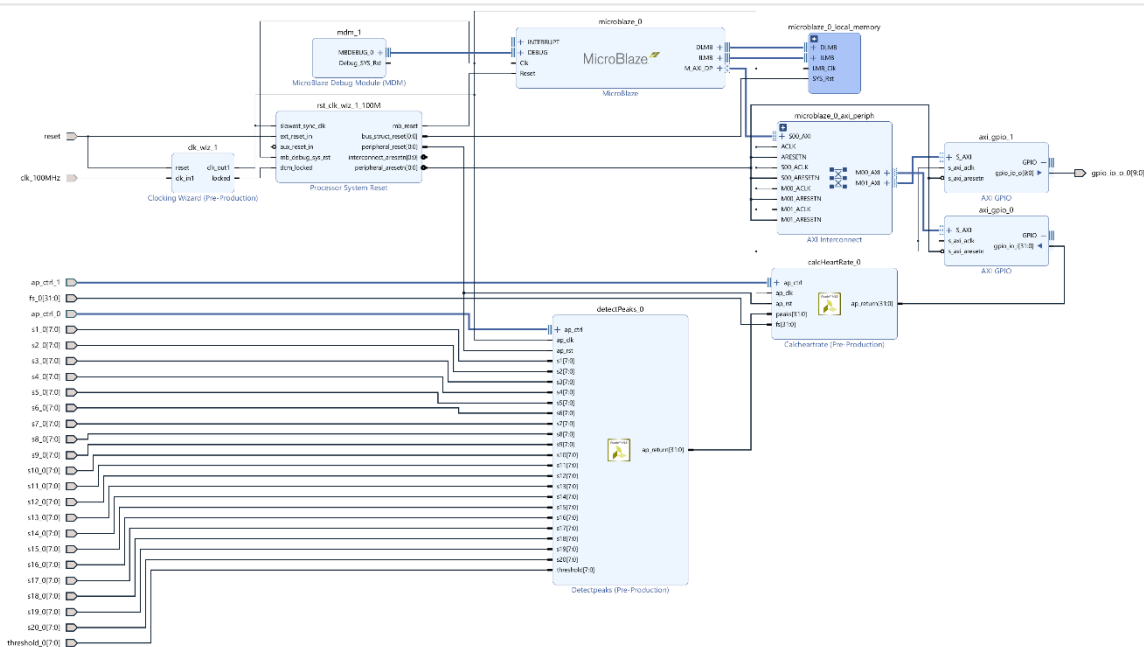
1. نمایش ضربان قلب

- آماده‌سازی پروژه و ساخت بلوک دیاگرام:

ابتدا مشابه قبل، بلوک MicroBlaze را به طراحی اضافه می‌کنیم و سیگنال کلاک ورودی را تنظیم می‌کنیم. سپس IP Core های تولید شده به کمک نرم‌افزار HLS را به طراحی اضافه می‌کنیم. یک GPIO برای ورودی در نظر گرفته و اندازه آن را 32 بیتی تعریف می‌کنیم، و یک GPIO دیگر به عرض 10 بیت را برای خروجی در نظر می‌گیریم (که شامل ورودی‌های سون‌سگمنت خواهد بود).

IP Core مربوط به تشخیص پیک‌ها، 20 ورودی دریافت می‌کند که مربوط به مقادیر سیگنال‌ها هستند. این ورودی‌ها را بعداً در تست‌بنچ ساخته شده، از فایل داده‌شده خواهیم خواند. همچنین فرکانس نمونه‌برداری، کلاک، سیگنال‌های کنترلی و ریست را نیز به عنوان ورودی به این بلوک می‌دهیم و در خروجی، آرایه 32 بیتی شامل اطلاعات پیک‌ها (تعداد و محل وقوع) را دریافت می‌کنیم.

سپس این خروجی همراه با فرکانس نمونه‌برداری، سیگنال‌های کلاک و ریست به عنوان ورودی به بلوک محاسبه ضربان قلب داده می‌شود. خروجی این بلوک همان مقدار ضربان قلب محاسبه شده‌است که باید بر روی سون‌سگمنت نمایش داده شود. این مقدار به عنوان ورودی از طریق پورت GPIO به میکرو داده شده و در آنجا از طریق کدی که در SDK نوشته خواهد شد، این عدد را تبدیل به یک سیگنال 10 بیتی برای ارسال به سون‌سگمنت می‌کند. نهایتاً از طریق پورت GPIO خروجی، این سیگنال محاسبه شده را به خروجی برمی‌گردانیم.



- توضیحات کد نوشته شده در SDK به زبان C:
همانگونه که پیش تر اشاره شد، این کد مربوط به تبدیل ضربان قلب محاسبه شده به سیگنال ورودی سون سگمنت است.
- ابتدا پورت های ورودی و خروجی GPIO را تعریف و Initialize می کنیم، و در خروجی مقدار اولیه 1 را می نویسیم (قسمت بنفش)
- سپس سه متغیر برای محاسبه رقم یکان، رقم دهگان و تعیین سون سگمنتی که باید روشن باشد تعریف می کنیم. (قسمت آبی)
- در حلقه while اصلی، ابتدا سیگنال ورودی را می خوانیم و آن را در متغیر 32 بیتی input ذخیره می کنیم. سپس برای محاسبه رقم دهگان، مقدار صحیح تقسیم این متغیر به 10 را محاسبه کرده و برای رقم یکان نیز باقی مانده تقسیم به 10 را محاسب می کنیم. (قسمت سبز)
- در ادامه بررسی می کنیم که سیگنال خروجی در این لحظه از اجرای کد برای سون سگمنت یکان ساخته می شود یا دهگان؛ و در هر صورت تابع ساخت این خروجی را فراخوانی کرده و سپس مقدار متغیر کنترلی را بین 0 و 1 toggle کرده و 2ms تاخیر ایجاد می کند. به این ترتیب پس از اجرای مجدد حلقه، خروجی برای سون سگمنت دیگر تولید خواهد شد. (قسمت زرد)
- نهایتاً تعریف تابع را مطابق با کد مربوط به شمارنده دو رقمی، بعد از تابع main می نویسیم (به علت تشابه، برای طولانی نشدن گزارش از نوشتن تمامی کیس ها خودداری شده است.)

```
#include "xparameters.h"
#include "xgpio.h"

#define INPUT_DEVICE_ID  XPAR_AXI_GPIO_0_DEVICE_ID
#define OUTPUT_DEVICE_ID XPAR_AXI_GPIO_1_DEVICE_ID

XGpio inputGpio, outputGpio;

int main()
{
    int inputStatus;
    int outputStatus;

    inputStatus = XGpio_Initialize(&inputGpio, INPUT_DEVICE_ID);
    outputStatus = XGpio_Initialize(&outputGpio, OUTPUT_DEVICE_ID);

    // Set the direction of the GPIO channels
    // Set input channel as input (32-bit width)
    XGpio_SetDataDirection(&inputGpio, 1, 0xFFFFFFFF);

    // Set output channel as output (10-bit width)
    XGpio_SetDataDirection(&outputGpio, 1, 0x00000000);

    XGpio_DiscreteWrite(&outputGpio, 1, 1);

    int ones = 0;
    int tens = 0;
    int control = 0;
```



```
while (1)
{
    u32 input = XGpio_DiscreteRead(&inputGpio, 1);

    ones = (int)(input % 10);
    tens = (int)(input / 10);
```

```
    if (control == 0)
    {
        seven_seg_update(ones, control);
        control = 1;
    }
    else
    {
        seven_seg_update(tens, control);
        control = 0;
    }

    MB_Sleep(1);
}

return XST_SUCCESS;
}
```

```
void seven_seg_update (int i, int control)
```

```
{
    if (control == 0)
    {
        switch (i)
        {
            case 0:

                XGpio_DiscreteWrite(&outputGpio, 1, 0b0111111100);
                break;

            case 1:
                XGpio_DiscreteWrite(&outputGpio, 1, 0b0101100000);
                break;

            case 2:
                XGpio_DiscreteWrite(&outputGpio, 1, 0b0111011010);
                break;

            .....
        }
    }
}
```

- ساخت Testbench و بررسی نتایج شبیه‌سازی:

در ابتدای فایل تست‌بنچ ساخته شده، component مربوط به تاپ ماژول را تعریف می‌کنیم؛ سپس سیگنال‌های مورد استفاده را تعریف کرده و port mapping را انجام می‌دهیم. پس از آن باید چند process تعریف کنیم که در ادامه هر یک بررسی خواهند شد:

- Clock Process

سیگنال کلاک را به اندازه نصف زمان پریود کلاک ورودی (100MHz) 0 کرده و در نیمه دیگر تناوب 1 می‌کنیم. به این ترتیب پالس کلاک ساخته خواهد شد.

```
--Clock process definitions
clk_process : process
begin

clk_100MHz <= '0;

wait for clk_period/2;

clk_100MHz <= '1;

wait for clk_period/2;

end process clk_process;
```

- Stimulus Process

در این بخش، بعد از گذشت 100ns سیگنال ریست را 0 کرده و به سیستم اجازه می‌دهیم تا دریافت ورودی و انجام محاسبات را آغاز کند.

```
--Stimulus process
stimulus_process : process
begin

reset <= '1;

wait for 100 ns;

reset <= '0;

wait;

end process stimulus_process;
```

- Control Process

در این بخش بعد از گذشت 100 ns، مقادیر threshold و فرکانس نمونه‌برداری را به ورودی بلوک Peak Detection می‌دهیم.

```
control_process : process
```

```

begin

threshold_0 <= (others => '0');

fs_0 <= (others => '0');

wait for 100 ns;

--Apply stimulus values

threshold_0 <= x"7F;"

fs_0 <= x"00000014;"

wait;

end process control_process;

```

Operation Process ○

در این قسمت ابتدا بعد از 100ns اولین بلوک را راه اندازی می کنیم. سپس تا زمان آماده شدن خروجی این بلوک صبر کرده و بلوک بعدی را راه اندازی می کنیم و ضمناً استارت بلوک های قبل را غیر فعال می کنیم.

```

--Start the operation
operation_process : process
begin

ap_ctrl_0_start <= '1;'

wait for 100 ns;

--Wait for the operation to complete
wait until ap_ctrl_0_done = '1;'

ap_ctrl_0_start <= '0;'

ap_ctrl_1_start <= '1;'

wait for 100 ns;

wait until ap_ctrl_1_done = '1;'

ap_ctrl_1_start <= '0;'

wait;

```

```
end process operation_process;
```

Read Process ○

در این بخش، به کمک روشی که در ویدیو معرفی شد فایل داده شده را سطر به سطر خوانده و مقادیر خوانده شده را در سیگنال‌هایی می‌ریزیم که به عنوان پایه ورودی به بلوک Peak Detection متصل هستند.

```
p_read : process(clk_100MHz, reset)

constant NUM_COL : integer := 20; -- number of column of file

type t_integer_array is array(integer range <> ) of integer;

file test_vector : text open read_mode is
"C:/Users/USER/Desktop/ECG.txt;"

variable row : line;

variable v_data_read : t_integer_array(1 to NUM_COL);

variable v_data_row_counter : integer := 0;

begin

if(falling_edge(reset)) then

--read from input file in "row" variable

if(not endfile(test_vector)) then

v_data_row_counter := v_data_row_counter + 1;

readline(test_vector,row);

end if;

--read integer number from "row" variable in integer array

for kk in 1 to NUM_COL loop

read(row,v_data_read(kk));

end loop;

s10_0 <= std_logic_vector(to_unsigned(v_data_read(10), 8));

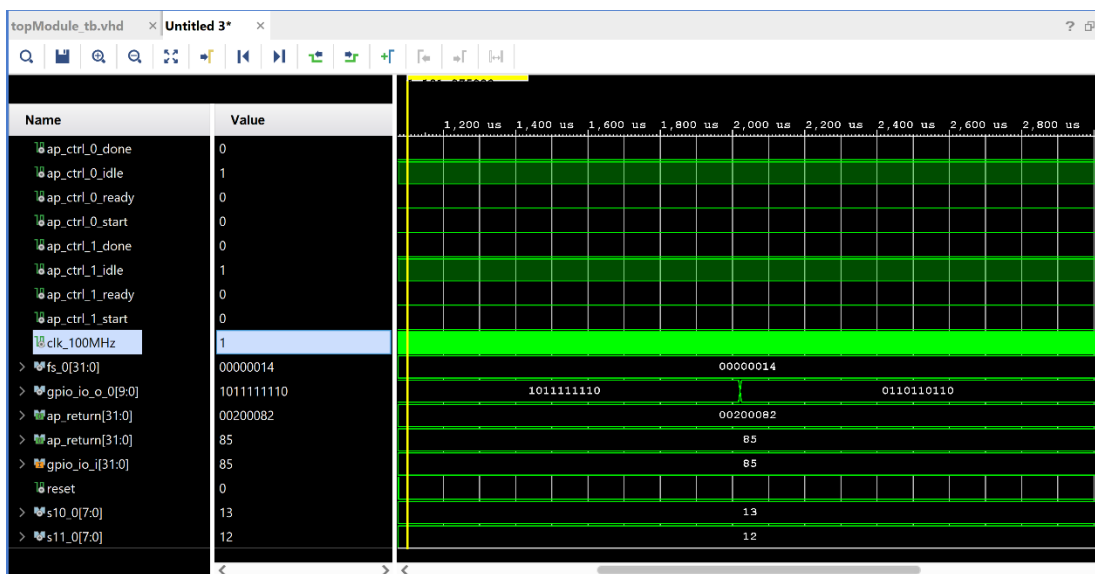
s1_0 <= std_logic_vector(to_unsigned(v_data_read(1), 8));

s2_0 <= std_logic_vector(to_unsigned(v_data_read(2), 8));
```

```
s3_0 <= std_logic_vector(to_unsigned(v_data_read(3), 8));
s4_0 <= std_logic_vector(to_unsigned(v_data_read(4), 8));
s5_0 <= std_logic_vector(to_unsigned(v_data_read(5), 8));

.....
```

نهایتاً فایل تست‌بنچ را اجرا کرده و موج‌های خروجی را بررسی می‌کنیم:



مشاهده می‌شود که خروجی دریافت شده از بلوک Peak Detection با خروجی که از تست‌بنچ نوشته شده در نرم‌افزار HLS بدست آمد برابر است. همچنین خروجی دریافت شده از بلوک calcHeartRate برابر با 85bpm است که مقدار مورد انتظار ماست. در ادامه، خروجی GPIO متناوباً مقادیر مربوط به رقم 8 (با بیت‌های کنترلی 10 که مربوط به دهگان است) و رقم 5 (با بیت‌های کنترلی 01 که مربوط به یکان است) را اتخاذ می‌کند. بنابراین، IP-Core های ساخته شده توسط نرم‌افزار به درستی کار می‌کنند.