پروژه درس مدارهای منطقی برنامه پذیر

استاد درس : دکتر شریعتمدار

آنوشا شریعتی - ۹۹۲۳۰۴۱

املین غازاریان _ ۹۹۲۳۰۵۶

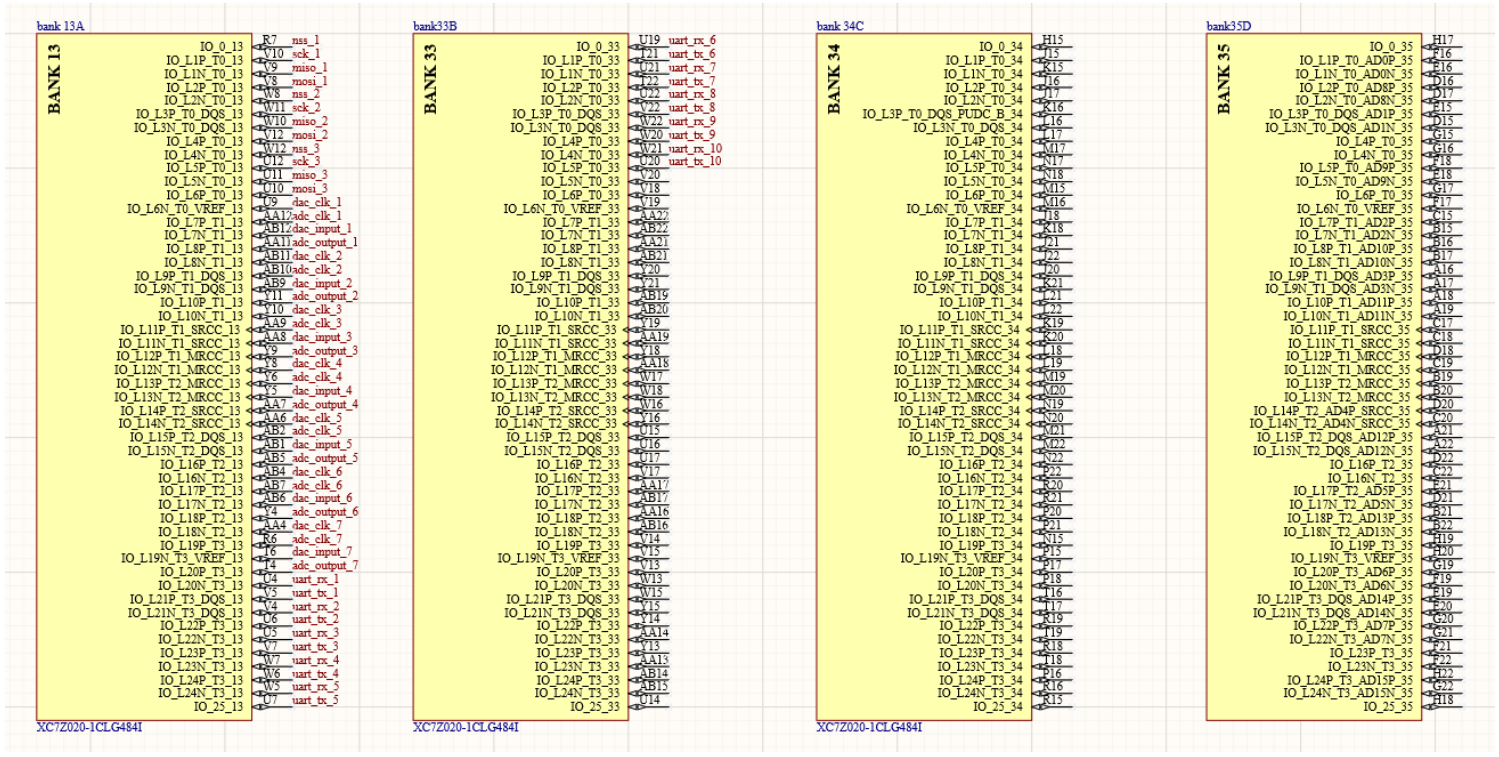
مهشاد اکبری سریزدی - ۹۹۲۳۰۹۳

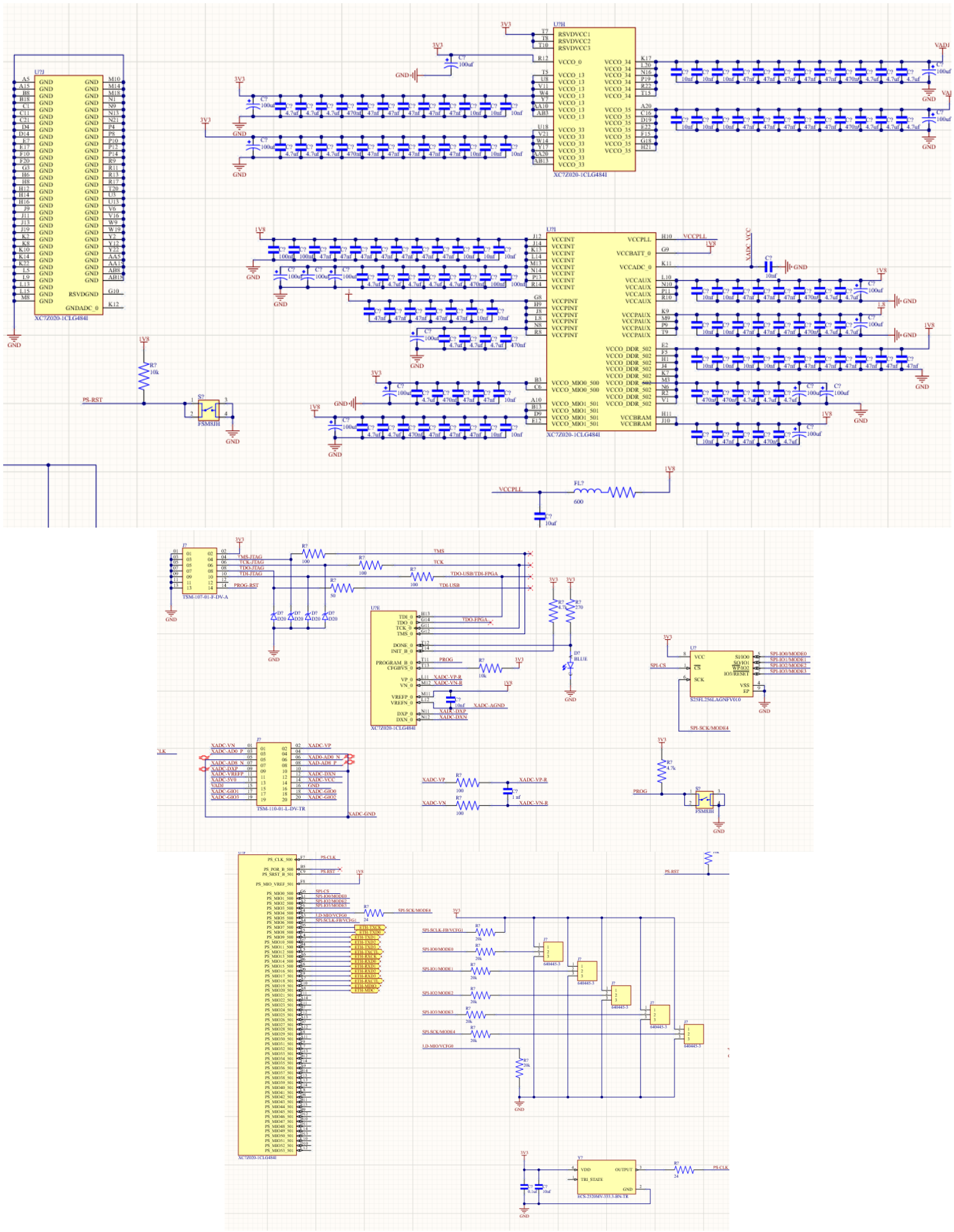
بخش اول :

۱-۱ : طراحی مفهومی :

- **هسته اصلی**

در طراحی مدار با توجه به داده های پروژه تصمیم بر این شد که از هسته fpga استفاده کنیم. به همین علت با استفاده از برنامه vivado به دنبال یک مدل مناسب و به صرفه گشتیم و تصمیم بر این شد که از مدل xcVz۰۲۰cig٤٨٤-۱ استفاده کنیم. این تراشه ۴۸۴ پین خروجی ورودی دارد و برای مسئله ما مناسب است. این هسته در Zed Board شرکت AVNET نیز مورد استفاده قرار گرفته است که در بخش های از پروژه از طراحی این بورد استفاده شده است. در ادامه با انتخاب لایبرری شماتیک و فوت پرینت مناسب مدار زیر را به شماتیک اضافه کرده و پایه های مورد نیاز را متصل میکنیم. مدار مربوط به تغذیه نیز مانند مدار طراحی شده در بورد zed board به صورت زیر قرار دادیم.





- **ورودی های آنالوگ**

طبق صورت پروژه ۱۰۰ ورودی آنالوگ داریم. برای پردازش این ورودی ها نیاز است آنها را به سیگنال دیجیتال تبدیل کنیم که برای این کار از ماژول ADC استفاده میکنیم. خروجی این ماژول را توسط پروتکل SPI به هسته FPGA متصل میکنیم. ماژول انتخابی ما برای این قسمت AD۵۵۹۰ است. علت این انتخاب این است که این ماژول هم ADC و هم DAC دارد و چون به هر دو این ماژول ها نیاز است انتخاب بهینه ای است. همچنین این ماژول ۱۶ ورودی میگیرد که مناسب مسئله ما که ورودی های زیادی دارد است. این ماژول دارای دو واحد SPI مجزاست که با یکی از هسته FPGA سیگنال دیجیتال را میگیرد و با واحد DAC به ۱۶ بیت خروجی آنالوگ تبدیل میکند و با واحد SPI دیگر که کلاک مجزا دارد خروجی واحد ADC را که سیگنالی دیجیتال است به FPGA میفرستد. اتصالات داخلی این ماژول مانند تصویر زیر است .

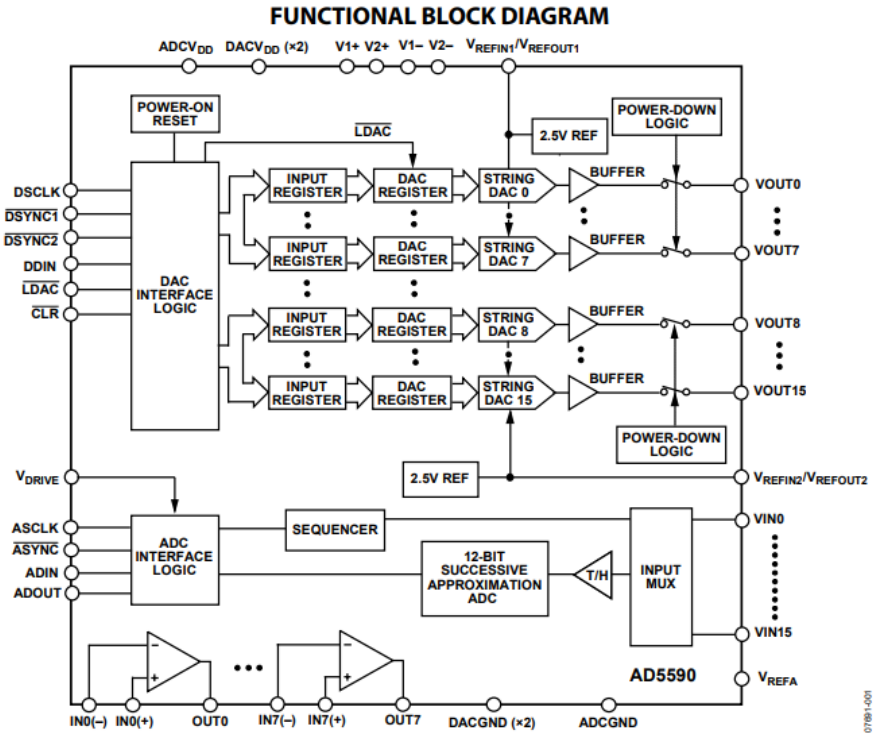
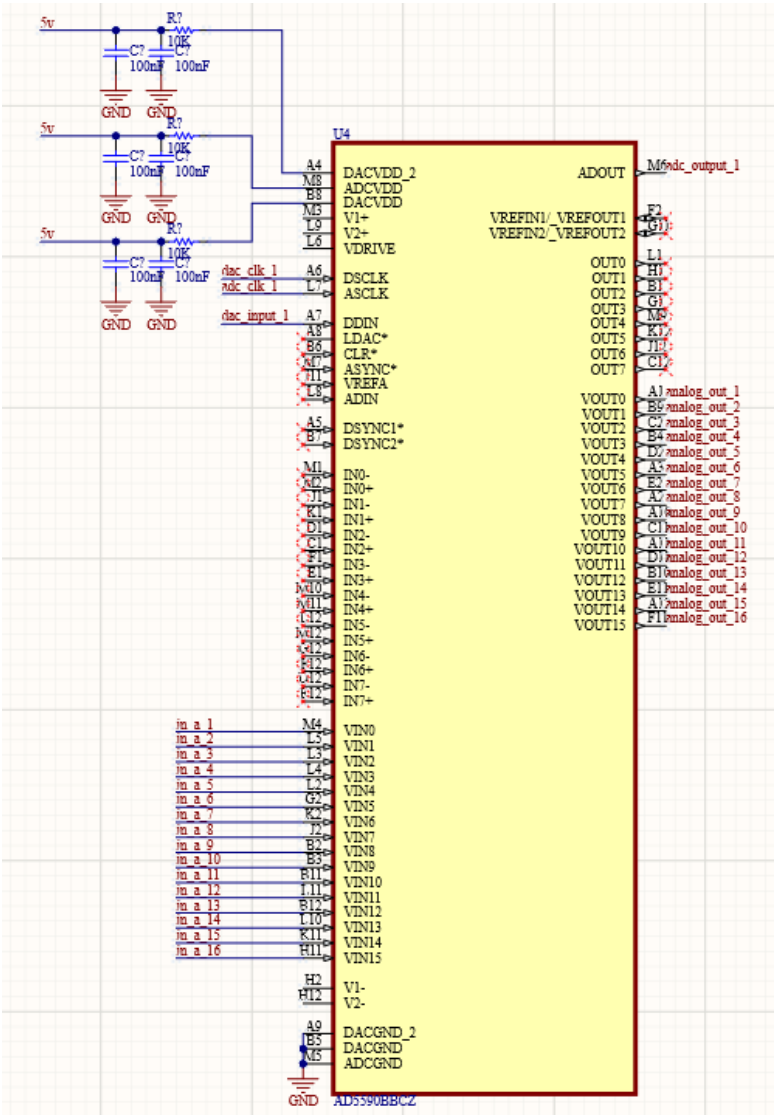


Figure 1.

با توجه به دیتاشیت و پین آوت قطعه شماتیک را به صورت زیر کشیده و از ۷ ماژول AD۵۵۹۰ برای ۱۰۰ خروجی و ورودی آنالوگ استفاده میشود همچنین پایه های مربوط به پروتکل SPI که شامل DDIN, DSCLK , ADOUT, ASCLK میشوند به FPGA متصل میشوند.



- **خروجی های آنالوگ**

برای طراحی قسمت مربوط به ۱۰۰ خروجی آنالوگ نیاز داشتیم سیگنال دیجیتال خروجی ار FPGA را به سیگنال آنالوگ تبدیل کنیم که به این منظور از ماژول DAC استفاده شد. ماژول DAC استفاده شده همان ماژول توضیح داده شده در بالا است.

با توجه به دیتاشیت این ماژول جریان پایه های آنالوگ خروجی حدود ۵۰ میلی آمپر جریان دارند که مناسب خواسته مسئله نیست به همین دلیل نیاز به استفاده از یک ماژول تقویت کننده جریان داریم. در طراحی این قسمت از مسئله از ماژول استفاده کردیم. مدار داخلی این ماژول به صورت زیر است و جریان پایه های آنالوگ خروجی را تا ۲۰۰ میلی امپر میرساند. همچنین پین آوت این ماژول به صورت زیر است.

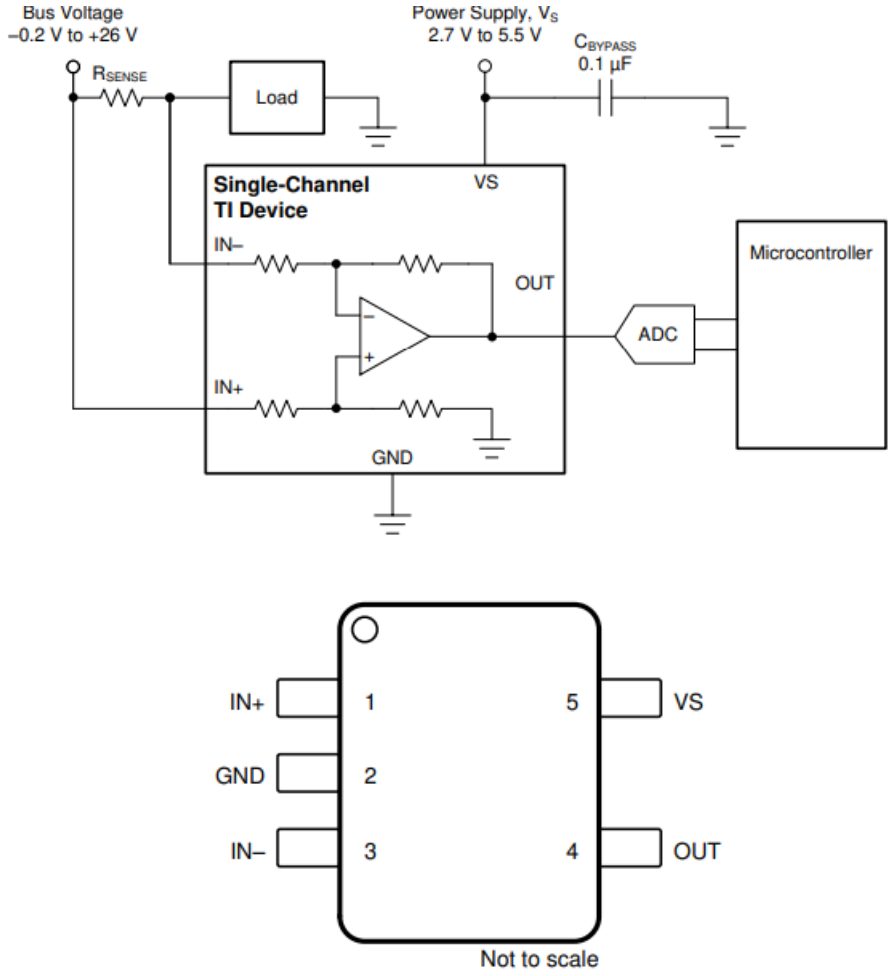
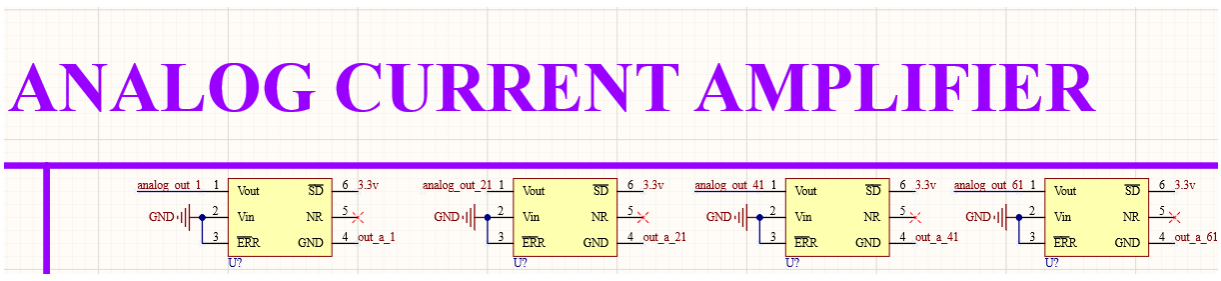
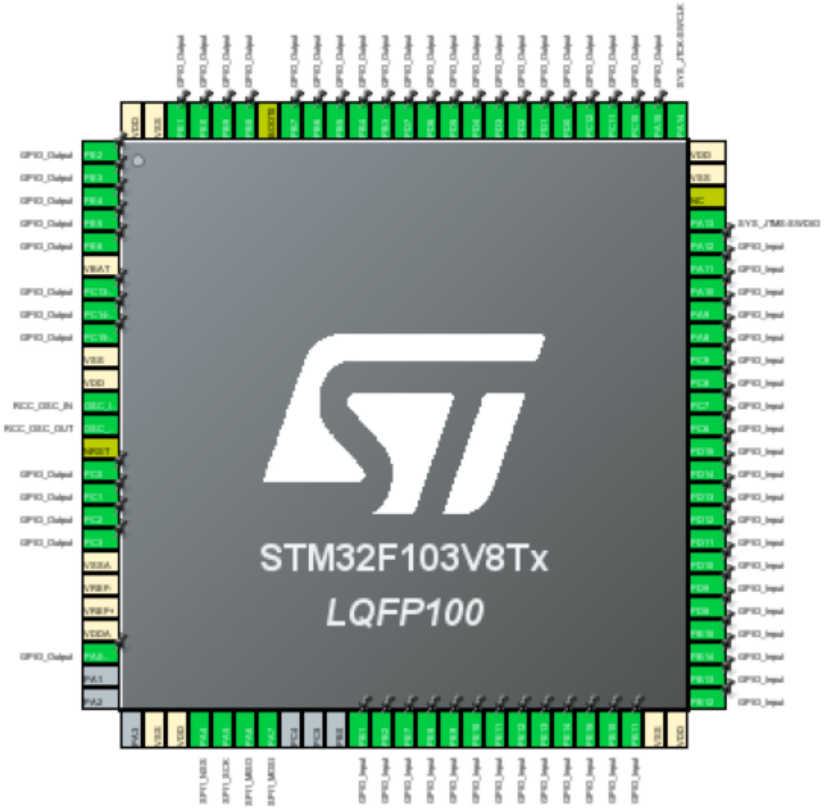


Figure 6-2. INA180: DBV Package 5-Pin SOT-23 (Pinout B) Top View



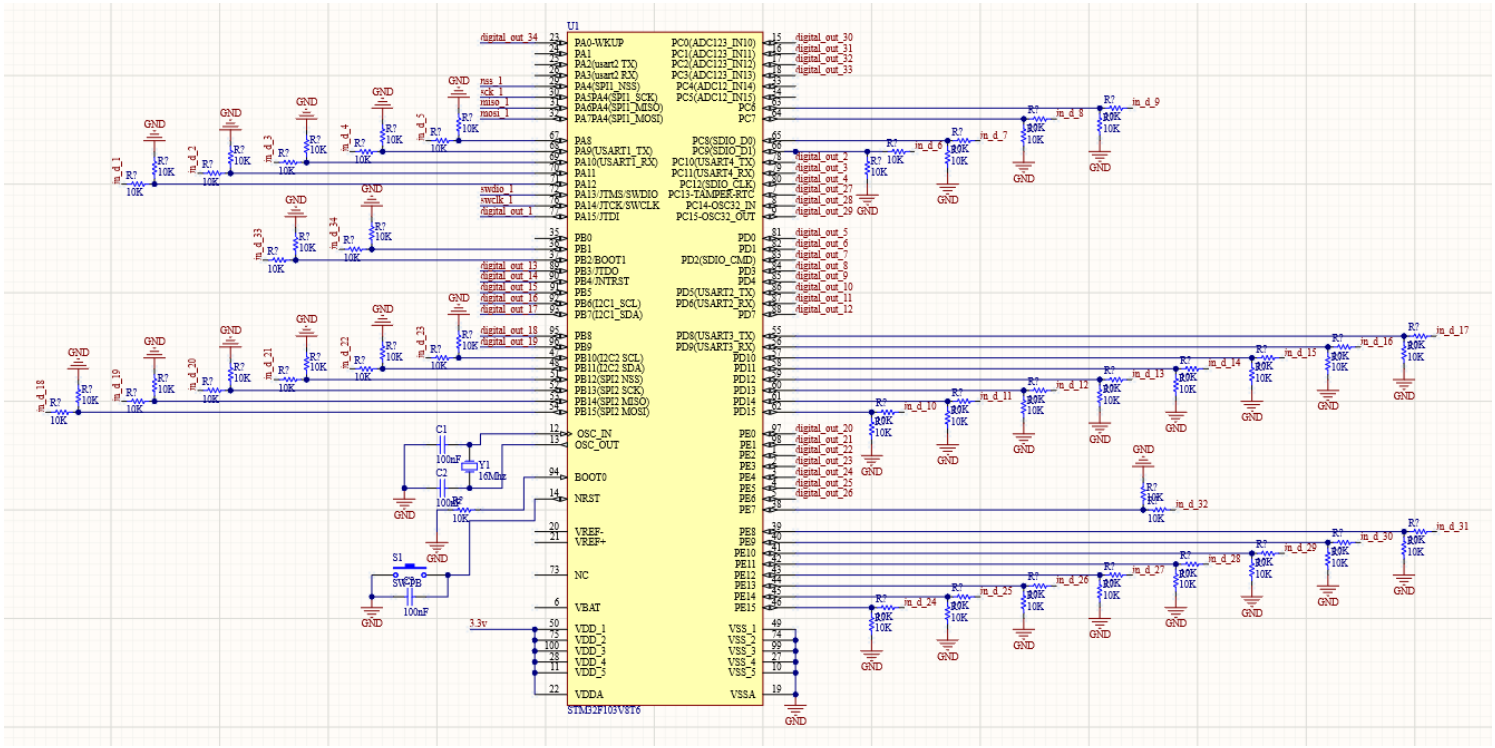
- **ورودی های دیجیتال**

در طراحی این قسمت نیاز داریم ۱۰۰ ورودی دیجیتال را پردازش کنیم. به این علت که تعداد پایه های خواسته شده در صورت پروژه خیلی زیاد است نمیتوان این ورودی ها را به صورت مستقیم به هسته FPGA متصل کرد. یکی از راه حل ها برای این مشکل استفاده از شیفت رجیستر است. با توجه به ۸ یا ۱۶ بیتی بودن این ماژول باز به تعداد زیادی از ان احتیاج پیدا میکنیم به همین خاطر سراغ راه حل دیگری میرویم. راه حل انتخاب شده استفاده از تعدادی میکرو کنترلر است که ورودی هارا به صورت مستقیم به آن وصل نموده و ارتباط بین میکرو ها و FPGA را با پروتکل SPI برقرار کنیم. علت استفاده از SPI راحتی و در دسترس بودن است. برای این قسمت از سه عدد میکروکنترلر ARMSTM۳۲F۱۰۳V۸T٦ استفاده کردیم. توجه شود که استفاده از میکرو کنترلر علاوه بر دلایل بالا یک نوع محافظت هم محسوب میشود. پایه های میکرو را مانند شکل زیر به ورودی و خروجی های مربوطه متصل میکنیم.



سطح ولتاژ ورودی های دیجیتال ۰ تا ۱۰ ولت است و ما میخواهیم به ۳.۳ ولت مورد استفاده میکروکنترلر برسد. به همین علت در ورودی های میکرو کنترلر با استفاده از دو مقاومت یک تقسیم ولتاژ انجام میدهیم.

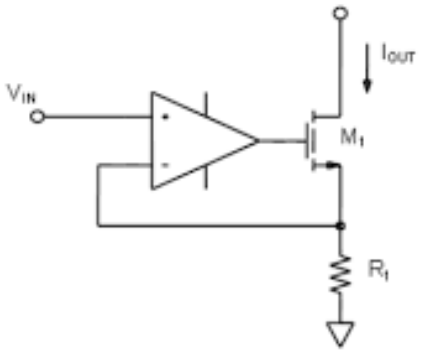
همچنین قطعات مورد نیاز میکروکنترلر اعم از خازن ها و مقاومت ها و کریستال و غیره را به آن متصل میکنیم. سپس پایه های پروتکل SPI را به FPGA متصل میکنیم.

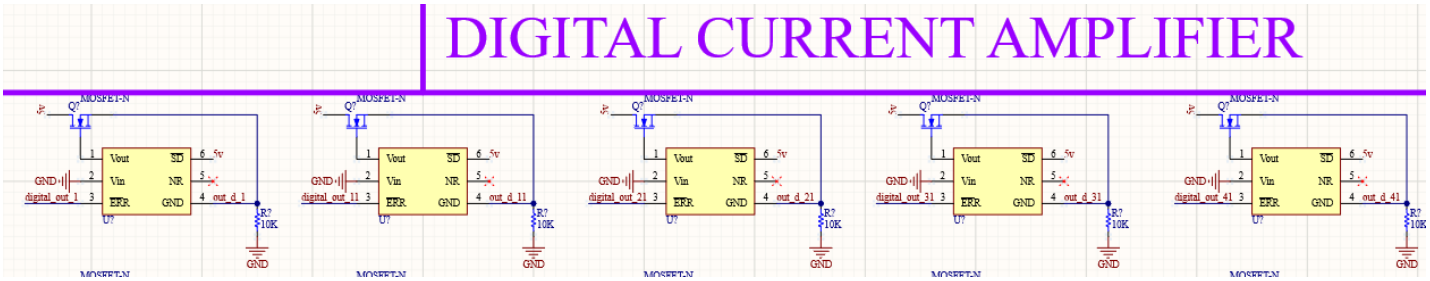


- **خروجی های دیجیتال**

در طراحی این قسمت ۱۰۰ سیگنال خروجی را میخواهیم خروجی بگیریم که جریان حدود ۲۰۰ میلی آمپر داشته باشند. همانطور که در قسمت قبل توضیح داده شد برای صرفه جویی در مصرف پایه ها از میکرو استفاده میکنیم. و چون حداکثر جریان خروجی میکرو حدود ۱۰ میلی آمپر است نیاز داریم جریان خروجی را تقویت کنیم و برای این کار از مدار زیر استفاده میکنیم

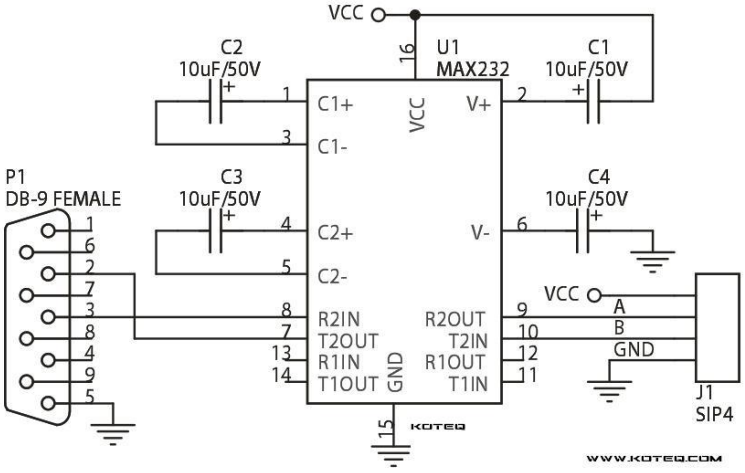
که از یک ماژول اپ امپLT٦٢٠٢ و یک ترانزیستور FDC٢٥١٢ و یک مقاومت تشکیل شده است.



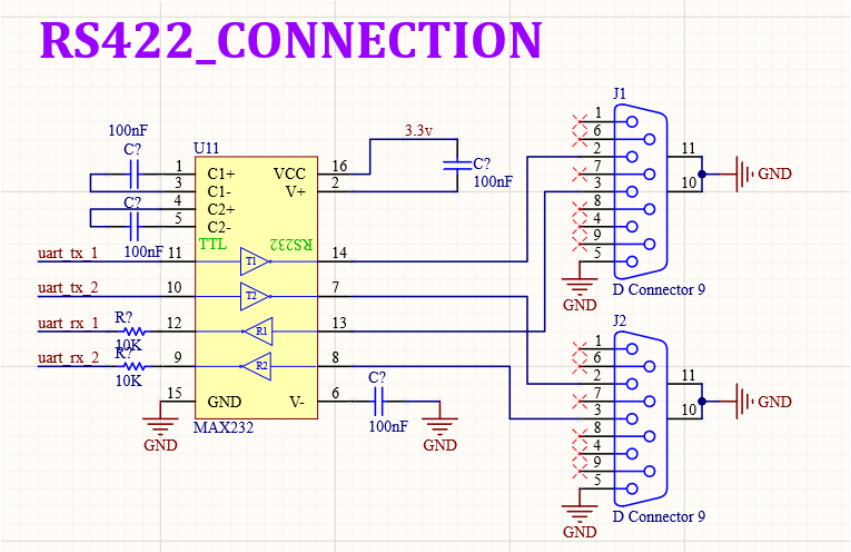


- **درایور RS٤٢٢**

یک ارتباط سریال را فراهم میکند که بر پایه پروتکل uart می باشد . شکل پین های ارتباطی آن به صورت زیر می باشد. توسط ماژول max٢٣٢ اتصالات rs٤٢٢ را فراهم کردیم . شکل ماژول در پایین آورده شده است .

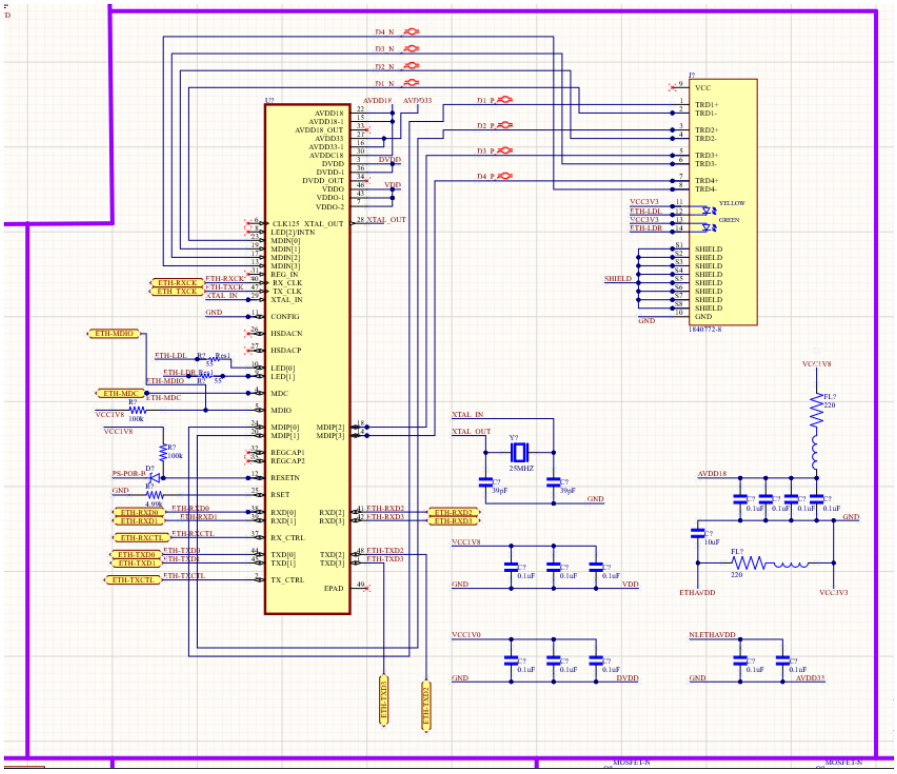


هر ماژول MAX۲۳۲ امکان برقراری دو ارتباط UART را میدهد به همین علت از ۵ ماژول به صورت زیر استفاده کردیم.



- **درایور ETHERNET**

برای ethernet از ماژول ethernet مربوط به خود fpga استفاده کردیم ، که شماتیک آن در سایت تولید کننده آن موجود بود و با توجه به آن شماتیک را رسم کردیم و فوت پرینت ها را اضافه کردیم.

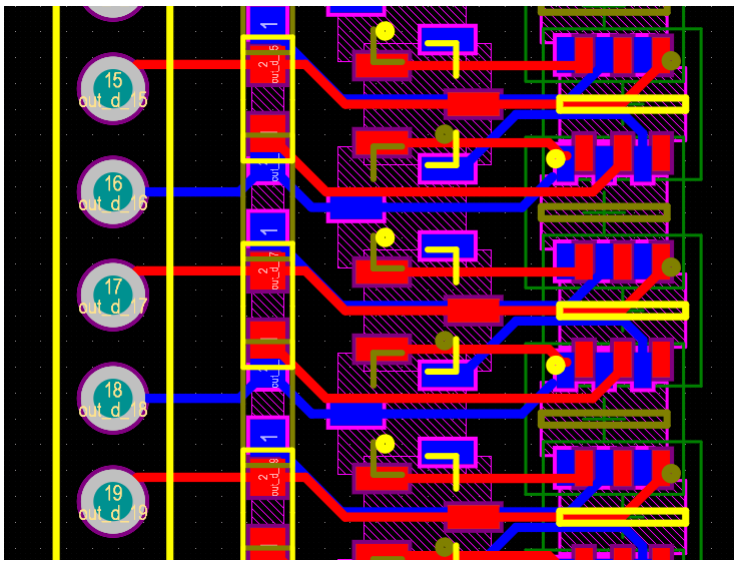


بخش دوم :

۲-۱ : قرار دهی قطعات

در این بخش با اطمینان از تکمیل بودن فوت پرینت های قطعات شماتیک را در فایل پی سی بی اضافه شده آپدیت کردیم و با برطرف کردن ارور ها قطعات را در جای مناسب قرار دادیم تا ترک کشی راحت تر شود. همچنین چون حجم پروژه بالا بود بهتر است از طراحی ۴ لایه استفاده کنیم و دو لایه میانی را زمین و VDD در نظر بگیریم تا ترک کشی راحت تر شود.

در قسمتی از طراحی مثلا برای تقویت کردن جریان دیجیتال خروجی از میکرو مدار را برای یک تقویت کننده رسم کرده و برای خروجی های دیگر از کپی اولی به صورت زیر استفاده میکنیم.



بخش سوم :

۳-۱: تعریف پین ها

برای fpga انتخابی مان فایل XDC مربوط به آن را در ویوادو در بخش constraints اضافه نمودیم.

در این جا با توجه به اتصالات top ماژولمان اتصالات سیگنال های ورودی در تاپ ماژول را به پین های fpga در هر بانک و با رعایت استاندارد ها و ولتاژ مربوط به هر بانک تعریف کردیم .

برای مثال در اینجا سیگنال GCLK که در تاپ ماژولمان هست را اتصالش به fpga را مشخص میکنیم . طبق manual این تراشه بخش clock configuration بر روی بانک ۱۳ اعمال میشود و بانک ۱۳ همواره ولتاژ ۳.۳ ولت را دارد .

پس با استفاده از دستور set_property در بخش get_ports نام سیگنال استفاده شده در تاپ ماژول را مینویسیم و با توجه به manual پین مربوط به تنظیمات کلاک Y۹ می باشد ، پس آن را مشخص میکنیم . با استفاده از دستور create_clock پریود کلاک برای سیگنال استفاده شده در تاپ ماژول را نیز محاسبه میکنیم . همچنین برای ۳.۳ ولت با دستور set property که در خط سوم هست استاندارد و مقدار ولتاژی بانک را مشخص میکنیم .

```
# Clock Source - Bank 13
# ------------------------------------------------------------------------
set_property PACKAGE_PIN Y9 [get_ports {GCLK}];  # "GCLK"
create_clock -name clk -period 10 [get_ports GCLK]
set_property IOSTANDARD LVCMOS33 [get_ports GCLK]
#
```

در بانک ۳۵ که بخش user switch می باشد ، سیگنال ریست تاپ ماژول را تعریف میکنیم .

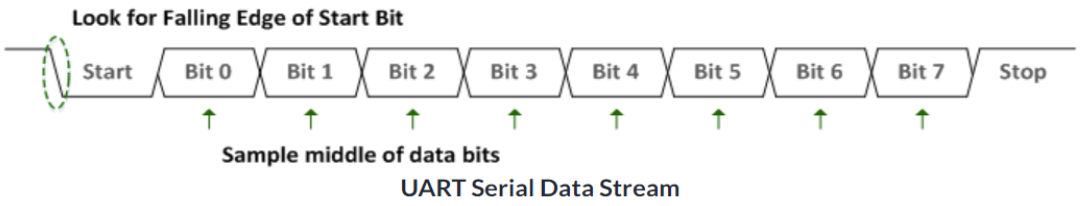
و پین مربوط به آن در fpga ، F۲۲ میباشد ، همچنین ولتاژ این بانک را روی ۱.۸ قرار می دهیم .در خط های پایین سیگنال s_۱ و s_۲ که در تاپ ماژول تعریف شده اند را به اتصالات پین های fpga مشخص میکنیم .

```
# User DIP Switches - Bank 35
# ------------------------------------------------------------------------
set_property PACKAGE_PIN F22 [get_ports {RESET}];  # "SW0"
set_property IOSTANDARD LVCMOS18 [get_ports RESET]

set_property PACKAGE_PIN G22 [get_ports {S_1}];  # "SW1"
set_property PACKAGE_PIN H22 [get_ports {S_2}];  # "SW2"
```

۳-۲ : کد نویسی RS٤٢٢

ابتدا برای انتخاب baud rate به دیتا شیت تراشه rs٤٢٢ نگاه می کنیم تا حداکثر سرعت آن را پیدا کنیم. سیستم های RS-٤٢٢ می توانند داده ها را با سرعت ۱۰ مگابیت بر ثانیه انتقال دهند پس baud rate می تواند ۹۶۰۰-۱۴۴۰۰-۱۹۲۰۰ و غیره باشد، ما مقدار ۱۹۲۰۰ را برای این پروژه انتخاب کردیم.

برای دریافت صحیح داده ها ، فرستنده و گیرنده باید در مورد نرخ باود توافق کنند. نرخ باود سرعتی است که داده ها با آن منتقل می شوند. مثال ۹۶۰۰ baud یعنی ۹۶۰۰ بیت در ثانیه FPGA به طور مداوم از خط نمونه برداری می کند. هنگامی خط انتقال داده UART از یک به صفر(از بالا به پایین) تغییر وضعیت می دهد ، می دانیم که یک کلمه داده UARTدر حال آمدن است. اولین تغییر وضعیت بیت شروع را نشان می دهد. هنگامی که ابتدای بیت شروع پیدا شد، FPGAتا وسط اولین بیت داده منتظر می ماند. این تضمین می کند که وسط بیت داده نمونه برداری می شود. از آن به بعد، FPGAفقط باید یک دوره بیت صبر کند (همانطور که توسط نرخ باود مشخص شده است) و بقیه داده ها را نمونه برداری کند. شکل زیر نحوه عملکرد گیرنده UART در داخل FPGA را نشان می دهد.



حال باید محاسبه کنیم که چقدر باید صبر کنیم تا به نصف بیت اول برسیم تا از آن نمونه برداری کنیم یعنی درواقع باید یک کانتر تعریف کنیم که تا نصف بیت بعدی بشمارد و بعد نمونه برداری کنیم.

$$number\ of\ clock\ cycles\ per\ bit = \frac{input\ clock\ frequency}{baud\ rate}$$
$$= \frac{۵۰MHz}{۱۹۲۰۰} = ۲۶۰۵$$

حال باید ماژول فرستنده و گیرنده را طراحی کنیم

گیرنده:

ابتدا ورودی و خروجی ها را تعریف کرده سپس سیگنال های مورد نیاز را تعریف می کنیم، همین طور استیت های استیت ماشین را تعریف می کنیم که در ادامه توضیح خواهیم داد.

```
entity UART_RX is
 generic (
    g_CLKS_PER_BIT : integer := 2605     -- represents the number of clock cycles per bit
    );
  port (
    i_Clk       : in  std_logic;
    i_RX_Serial : in  std_logic;
    o_RX_DV     : out std_logic;
    o_RX_Byte   : out std_logic_vector(7 downto 0)
    );
end UART_RX;

architecture Behavioral of UART_RX is

type t_SM_Main is (s_Idle, s_RX_Start_Bit, s_RX_Data_Bits,
                   s_RX_Stop_Bit, s_Cleanup);
  signal r_SM_Main : t_SM_Main := s_Idle;

  signal r_RX_Data_R : std_logic := '0';
  signal r_RX_Data   : std_logic := '0';

  signal r_Clk_Count : integer range 0 to g_CLKS_PER_BIT-1 := 0; --keep track of clock cycles
  signal r_Bit_Index : integer range 0 to 7 := 0;  -- 8 Bits Total - keep track of the current bit position.
  signal r_RX_Byte   : std_logic_vector(7 downto 0) := (others => '0'); --store the received byte
  signal r_RX_DV     : std_logic := '0'; --validity of the received data
```

در اینجا یک process حساس به لبه بالا رونده داریم که در ادامه استیت هایمان را تعریف می کنیم. در استیت idle متغییر هایمان را initialize می کنیم و اگر ورودی صفر شد یعنی بیت شروع تشخیص داده شده پس باید به استیت بیت شروع برود.

```
-- Purpose: Control RX state machine
p_UART_RX : process (i_Clk)
begin
  if rising_edge(i_Clk) then
    case r_SM_Main is
      when s_Idle =>
        r_RX_DV     <= '0';
        r_Clk_Count <= 0;
        r_Bit_Index <= 0;

        if r_RX_Data = '0' then       -- Start bit detected
          r_SM_Main <= s_RX_Start_Bit;
        else
          r_SM_Main <= s_Idle;
        end if;
```

سپس در استیت بیت شروع وسط بیت شروع را چک می کنیم تا مطمعن شویم هنوز یک می باشد و آن صفری که اول دریافت کردیم نویز نبوده. پس کانتر تا وسط آن مقداری که بالاتر حساب کرده بودیم باید بشمرد و وقتی کانتر به آن مقدار رسید (اگر نرسیده بود یکی به مقدار کانتر اضافه می شود و دوباره وارد این استیت می شود) چک می شود که داده صفر است یا یک، اگر صفر بود پس دیگه قطعا بیت شروع تشخیص داده شده و به استیت دریافت داده سریال می رود و در غیر این صورت به استیت idle بر می گردد تا دوباره بیت شروع تشخیص داده شود.

```
      -- Check middle of start bit to make sure it's still low
      when s_RX_Start_Bit =>

      -- waits for the middle of the start bit to ensure proper sampling
      if r_Clk_Count = (g_CLKS_PER_BIT-1)/2 then
        if r_RX_Data = '0' then
          r_Clk_Count <= 0;  -- reset counter since we found the middle
          r_SM_Main   <= s_RX_Data_Bits;
        else
          r_SM_Main   <= s_Idle;
        end if;
      else
        r_Clk_Count <= r_Clk_Count + 1;
        r_SM_Main   <= s_RX_Start_Bit;
      end if;
```

در استیت دریافت داده سریال این دریافت زمانی صورت می گیرد که مقدار کانتر به آن constant که محاسبه کردیم رسیده باشد و در غیر این صورت کانتر به شمارش ادامه می دهد. از آنجایی که ۸ بیت داده داریم ارسال میکنیم پس r_bit_index هر بار یکی اضافه می شود تا بیت بعدی دریافت شود و زمانی که مقدار این رجیستر از ۷ بیشتر شد یعنی ۸ بیت ارسال شده و باید برویم به استیت بیت پایان تا آن را تشخیص دهیم تا دریافت ما تمام شود.

```
-- Wait g_CLKS_PER_BIT-1 clock cycles to sample serial data
when s_RX_Data_Bits =>
  if r_Clk_Count < g_CLKS_PER_BIT-1 then
    r_Clk_Count <= r_Clk_Count + 1;
    r_SM_Main   <= s_RX_Data_Bits;
  else
    r_Clk_Count            <= 0;
    r_RX_Byte(r_Bit_Index) <= r_RX_Data;

    -- Check if we have sent out all bits
    if r_Bit_Index < 7 then
      r_Bit_Index <= r_Bit_Index + 1;
      r_SM_Main   <= s_RX_Data_Bits;
    else
    --meaning data bits have been finished
      r_Bit_Index <= 0;
      r_SM_Main   <= s_RX_Stop_Bit;
    end if;
  end if;
```

در استیت بیت پایان، این بیت را تشخیص داده و وقتی این بیت تشخیص داده شد سیگنال r_rx_dv یک می شود به این معنی که دیتا دریافتی valid بوده است سپس به استیت اخر می رویم که در واقع همان استیت reset است که مقدار سیگنال ها را صفر می کند و اماده می کند برای دریافت داده های سریال بعدی

```
-- Receive Stop bit. Stop bit = 1
when s_RX_Stop_Bit =>
  -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
  if r_Clk_Count < g_CLKS_PER_BIT-1 then
    r_Clk_Count <= r_Clk_Count + 1;
    r_SM_Main   <= s_RX_Stop_Bit;
  else
  --  It waits for the stop bit to finish and then asserts the data valid signal
    r_RX_DV     <= '1';
    r_Clk_Count <= 0;
    r_SM_Main   <= s_Cleanup;
  end if;

-- Stay here 1 clock
when s_Cleanup =>
  r_SM_Main <= s_Idle;
  r_RX_DV   <= '0'; when others =>
  r_SM_Main <= s_Idle;
```

فرستنده:

ابتدا ورودی و خروجی ها را تعریف کرده سپس سیگنال های مورد نیاز را تعریف می کنیم، همین طور استیت های استیت ماشین را تعریف می کنیم که در ادامه توضیح خواهیم داد.

```
entity UART_TX is
generic (
    g_CLKS_PER_BIT : integer := 2605
    );
  port (
    i_Clk       : in  std_logic;
    i_TX_DV     : in  std_logic;
    i_TX_Byte   : in  std_logic_vector(7 downto 0);
    o_TX_Active : out std_logic;
    o_TX_Serial : out std_logic;
    o_TX_Done   : out std_logic
    );
end UART_TX;

architecture Behavioral of UART_TX is

type t_SM_Main is (s_Idle, s_TX_Start_Bit, s_TX_Data_Bits,
                   s_TX_Stop_Bit, s_Cleanup);
  signal r_SM_Main : t_SM_Main := s_Idle;

  signal r_Clk_Count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal r_Bit_Index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal r_TX_Data   : std_logic_vector(7 downto 0) := (others => '0');
  signal r_TX_Done   : std_logic := '0';
```

در استیت idle مشابه قسمت قبل سیگنال ها را initialize می کنیم. حال اگر i_TX_DV یک شده باشد (نشان دهنده داده های معتبر برای انتقال است)، برای شروع انتقال به s_TX_Start_Bit منتقل می شود.

```
  p_UART_TX : process (i_Clk)
   begin
     if rising_edge(i_Clk) then

       case r_SM_Main is

         when s_Idle =>
           o_TX_Active <= '0';
           o_TX_Serial <= '1';         -- Drive Line High for Idle
           r_TX_Done   <= '0';
           r_Clk_Count <= 0;
           r_Bit_Index <= 0;

           if i_TX_DV = '1' then
             r_TX_Data <= i_TX_Byte;
             r_SM_Main <= s_TX_Start_Bit;
           else
             r_SM_Main <= s_Idle;
           end if;
```

در این استیت تا آن constant محاسبه شده صبر می کند تا بیت شروع به پایان برسد تا به استیت انتقال داده برود.

```
-- Send out Start Bit. Start bit = 0
when s_TX_Start_Bit =>
  o_TX_Active <= '1';
  o_TX_Serial <= '0';

  -- Wait g_CLKS_PER_BIT-1 clock cycles for start bit to finish
  if r_Clk_Count < g_CLKS_PER_BIT-1 then
    r_Clk_Count <= r_Clk_Count + 1;
    r_SM_Main   <= s_TX_Start_Bit;
  else
    r_Clk_Count <= 0;
    r_SM_Main   <= s_TX_Data_Bits;
  end if;
```

به اندازع g_CLKS_PER_BIT-۱ سیکل کلاک منتظر می ماند برای هر بیت داده تا تمام شود و برود بیت بعدی. و ادامه توضیحات مشابه قسمت قبلی می باشد.

```
-- Wait g_CLKS_PER_BIT-1 clock cycles for data bits to finish
when s_TX_Data_Bits =>
  o_TX_Serial <= r_TX_Data(r_Bit_Index);

  if r_Clk_Count < g_CLKS_PER_BIT-1 then
    r_Clk_Count <= r_Clk_Count + 1;
    r_SM_Main   <= s_TX_Data_Bits;
  else
    r_Clk_Count <= 0;

    -- Check if we have sent out all bits
    if r_Bit_Index < 7 then
      r_Bit_Index <= r_Bit_Index + 1;
      r_SM_Main   <= s_TX_Data_Bits;
    else
      r_Bit_Index <= 0;
      r_SM_Main   <= s_TX_Stop_Bit;
    end if;
  end if;
```

در استیت بیت پایان نیز این بیت تشخیص داده می ود و به اندازه g_CLKS_PER_BIT-۱ صبر می کند تا تمام شود و به استیت اخر برود و سیگنال ها ریست شوند و اماده درسافت داده های سریال بعدی باشد.

```
      -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
      if r_Clk_Count < g_CLKS_PER_BIT-1 then
        r_Clk_Count <= r_Clk_Count + 1;
        r_SM_Main   <= s_TX_Stop_Bit;
      else
        r_TX_Done   <= '1';
        r_Clk_Count <= 0;
        r_SM_Main   <= s_Cleanup;
      end if;

    -- Stay here 1 clock
    when s_Cleanup =>
      o_TX_Active <= '0';
      r_TX_Done   <= '1';
      r_SM_Main   <= s_Idle;

    when others =>
      r_SM_Main <= s_Idle;
```

تست بنچ:

ابتدا یک روش UART_WRITE_BYTE برای شبیه سازی انتقال یک بایت تعریف شده است سپس با تنظیم سیگنال خروجی o_serial بر روی '۰' انتقال بیت شروع را شبیه سازی می کند. دستور انتظار برای c_BIT_PERIOD نشان دهنده مدت زمان یک بیت دوره است و تاخیری را برای شبیه سازی زمان لازم برای ارسال بیت شروع ارائه می کند. قسمت بعدی کد از یک حلقه برای تکرار بر روی هر بیت (از بیت ۰ تا بیت ۷) از بایت داده ورودی i_data_in استفاده می کند. برای هر بیت، مقدار متناظر به سیگنال خروجی o_serial اختصاص داده می شود و انتقال هر بیت داده را شبیه سازی می کند. باز هم، دستور انتظار برای c_BIT_PERIOD تاخیرهایی را برای شبیه سازی زمان لازم برای انتقال هر بیت داده معرفی می کند. پس از ارسال تمام بیت های داده، رویه سیگنال خروجی o_serial را روی '۱' تنظیم می کند و انتقال بیت توقف را شبیه سازی می کند. انتظار نهایی برای دستور c_BIT_PERIOD یک تاخیر را برای شبیه سازی زمان لازم برای ارسال بیت توقف معرفی می کند.

```
  -- Low-level byte-write
  procedure UART_WRITE_BYTE (
    i_data_in       : in  std_logic_vector(7 downto 0);
    signal o_serial : out std_logic) is

begin

  -- Send Start Bit
    o_serial <= '0';
    wait for c_BIT_PERIOD;

    -- Send Data Byte
    for ii in 0 to 7 loop
      o_serial <= i_data_in(ii);
      wait for c_BIT_PERIOD;
    end loop;  -- ii

    -- Send Stop Bit
    o_serial <= '1';
    wait for c_BIT_PERIOD;
    end UART_WRITE_BYTE;
```

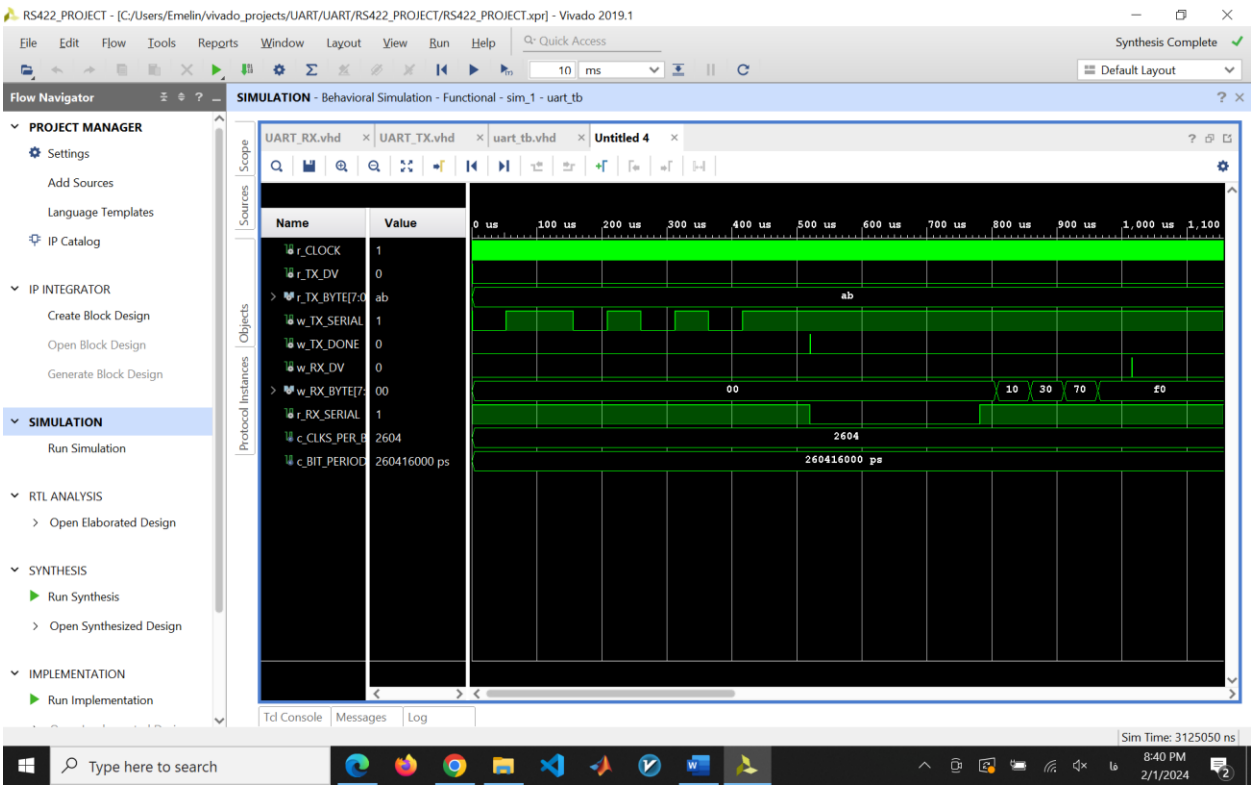
در این قسمت ماژول های فرستنده و گیرنده را طراحی کردیم را پورت مپ می کنیم.

```
begin

-- Instantiate UART transmitter
UART_TX_INST : uart_tx
  generic map (
    g_CLKS_PER_BIT => c_CLKS_PER_BIT
    )
    port map (
      i_clk       => r_CLOCK,
      i_tx_dv     => r_TX_DV,
      i_tx_byte   => r_TX_BYTE,
      o_tx_active => open,
      o_tx_serial => w_TX_SERIAL,
      o_tx_done   => w_TX_DONE
      );

  -- Instantiate UART Receiver
  UART_RX_INST : uart_rx
    generic map (
      g_CLKS_PER_BIT => c_CLKS_PER_BIT
      )
    port map (
      i_clk       => r_CLOCK,
      i_rx_serial => r_RX_SERIAL,
      o_rx_dv     => w_RX_DV,
      o_rx_byte   => w_RX_BYTE
      );
```

این بخش به UART که دستور ارسال یک بایت را شبیه سازی می کند. این قسمت ارسال یک فرمان (x”ab”) به UART را پس از لبه کلاک شبیه سازی می کند. روال UART_WRITE_BYTE برای شبیه سازی فرآیند نوشتن یک بایت در UART فراخوانی می شود. در نهایت، بایت دریافتی (w_RX_BYTE) بررسی می شود.

```
process is
begin

  -- Tell the UART to send a command.
  wait until rising_edge(r_CLOCK);
  wait until rising_edge(r_CLOCK);
  r_TX_DV   <= '1';
  r_TX_BYTE <= X"AB";
  wait until rising_edge(r_CLOCK);
  r_TX_DV   <= '0';
  wait until w_TX_DONE = '1';


  -- Send a command to the UART
  wait until rising_edge(r_CLOCK);
  UART_WRITE_BYTE(X"3F", r_RX_SERIAL);
  wait until rising_edge(r_CLOCK);

  -- Check that the correct command was received
  if w_RX_BYTE = X"3F" then
    report "Test Passed - Correct Byte Received" severity note;
  else
    report "Test Failed - Incorrect Byte Received" severity note;
  end if;

  assert false report "Tests Complete" severity failure;
```

نتایج:



تست بنچ به گونه ای است که به صورت جدا قسمت فرستنده و گیرنده تست می شود. همان طور که مشاهده می کنیم فرمانی که ارسال کردیم به درستی ارسال شده (x”ab”) و فرمانی که دریافت کنیم نیز بیت به بیت به درستی دریافت شده تا به عدد نهایی می رسد.

منابع:

منبع استفاده شده : https://nandland.com/uart-serial-port-module/

فایل RS٤٢٢_PROJECT پروژه اصلی می باشد.

فایل rs٤٢٢_proj کدی هست که با استفاده از ماژول فیلتر ورودی زده شده.(تکمیلی)

توضیحات تکمیلی:

برای قسمت گیرنده می توانیم یک ماژول فیلتر ورودی نیز قرار دهیم تا نویز های کوتاه و گذرا در ورودی را فیلتر کند.

در این ماژول دو پارامتر داریم)که در ادامه کاربرد انها را خواهیم دید(و یک رجیستر count ، که یک شمارنده ۸ بیتی است که تعداد ۱ های متوالی را در سیگنال ورودی می شمارد.

```
entity input_filter is
Port (
    Clk  : in  STD_LOGIC;
    DIn  : in  STD_LOGIC;
    DOut : out STD_LOGIC
  );
end input_filter;

architecture Behavioral of input_filter is

constant bit_num : integer := 2;
  constant count_m : integer := 7;

  signal count : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
```

در ادامه کد بالا را داریم که یک بلوک always است و با لبه بالا رونده کلاک فعال می شود، که در داخل این بلوک چک می شود که مقدار Din برابر یک است یا صفر. اگر برابر یک بود و مقدار

m_countبیشتر از count بود یکی به شمارنده اضافه می شود و اگر برابر صفر بود و مقدار countاز ۲ بیشتر بود یکی از شمارنده کم می شود. در نهایت مقدار num_bit[count]را در داخل dout می ریزیم.

منطق و هدف پشت این کد این است که، پارامتر m_count دارای مقدار ۷ است که درواقع یک مقدار آستانه است به طوری که اگر شمارنده ما(count)به این مقدار آستانه برسد یا بگذرد، به این معنی است که سیگنال ورودی به طور مداوم برای مدت زمان مشخصی (که این مدت زمان را مقدار استانه تعیین می کند) یک بوده است، و سپس اجازه می دهد تا خروجی DOoutبر اساس بیت انتخاب شده از count توسط num_bit ، تغییر کند. حال وقتی که سیگنال ورودی یک است و count از m_count کمتر باشد، این بدان معناست که سیگنال ورودی برای مدت زمان کافی برای رسیدن به آستانه بالا نبوده است تا dout براساس آن تغییر کند. در اینجا با افزایش countمقدار ۱های متوالی را می شماریم تا بدانیم چه مدت سیگنال ورودی یک بوده است تا دوباره در حلقه if مقدار count با مقدار m_countمقایسه شود. در اصل، این ماژول به گونه ای طراحی شده است که سیگنال های کوتاه و گذرا در ورودی را براساس منطقی که در بالا توضیح دادم، فیلتر کند.

```
begin
  process (Clk)
  begin
    if rising_edge(Clk) then
      if DIn = '1' then
        if count < count_m then
          count <= count + 1;
        end if;
      elsif DIn = '0' then
        if count > 2 then
          count <= count - 1;
        end if;
      end if;

      DOut <= count(bit_num);
    end if;
  end process;
```