به نام خدا

گزارش آزمایشگاه FPGA

آزمایش اول

گروه سه

علی ایمانقلی

میلاد عظیمی فشی

زهرا کشاورزهدایتی

فاطمه رحیمی

فهرست

Async_transmitter

```
module async_transmitter(
    input clk,
    input TxD_start,
    input [7:0] TxD_data,
    output TxD,
    output TxD_busy
);
```

- مقادیر پارامترها

```
parameter ClkFrequency = 50000000;
parameter Baud = 115200;
parameter Oversampling = 1;
```

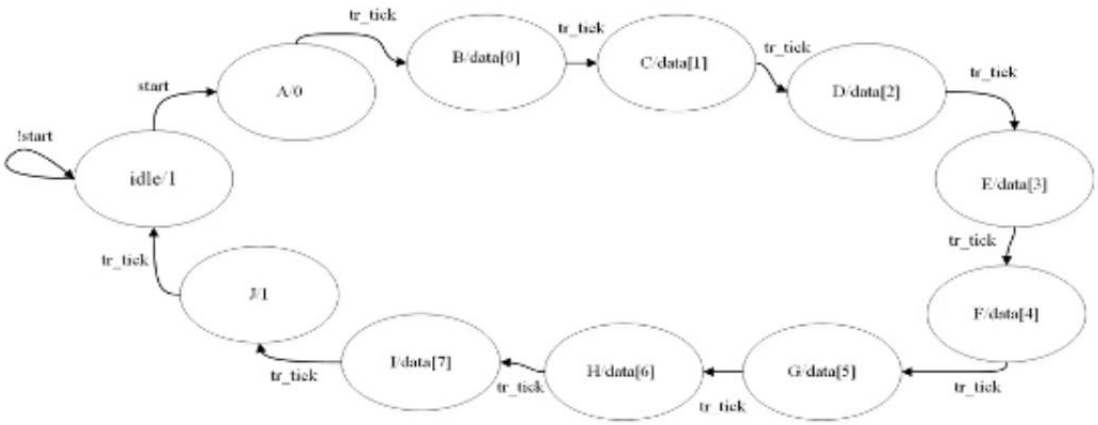
- گرفتن اینستنس از ماژول baud rate generator

زمانی که در حال ارسال داده باشد BaudTickGen فعال میشود (زمانی که در حال ارسال داده است busy یک میشود و enable که active low است به این شرط حساس است).

خروجی این ماژول هم به sample وصل شده است.

```
wire TxDTick;
BaudTickGen #(ClkFrequency, Baud, Oversampling) tickgen(.clk(clk), .enable(!TxD_busy), .tick(TxDTick));
```

- state machine



همانطور که در شکل دیده میشود 11 استیت داریم که در بدنه always زیر پیاده سازی شده است. اگر start یک شده باشد و در حال ارسال داده دیگری نباشیم (ready که در صورت بودن در استیت 0000 یک میشود، هم یک باشد)، هشت بیت ورودی را دریافت کرده و در شیفت رجیستر قرار میدهیم. بعد از این هر بار به اندازه یک بار tick زدن (که با یک شدن TxDTick مشخص میشود) صبر میکنیم و یک بیت از دیتا را ارسال میکنیم (شیفت رجیستر را یک بار شیفت میدهیم و وارد استیت بعد میشویم).

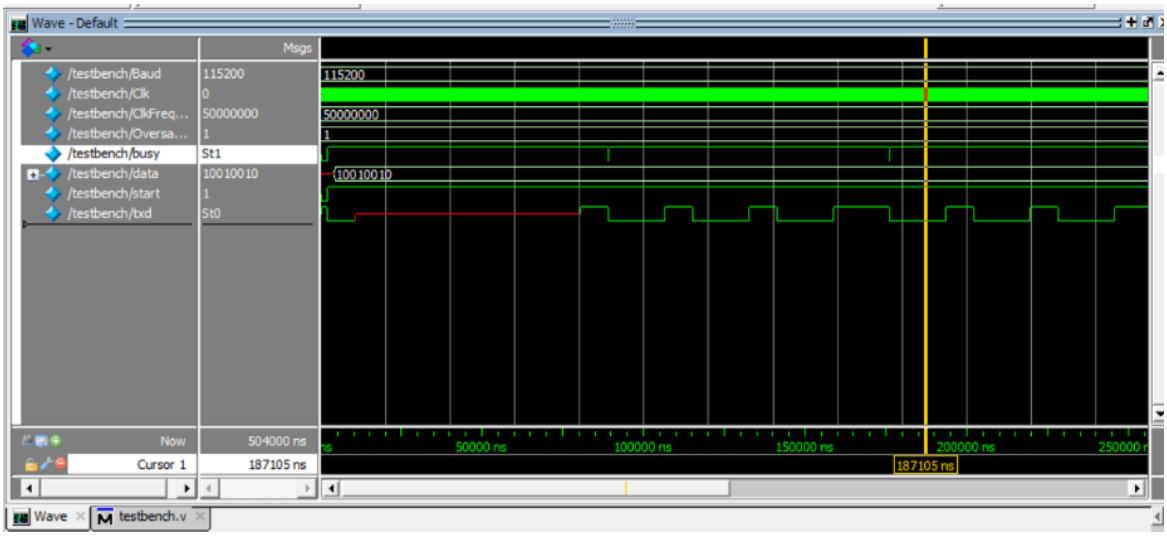
```
reg [3:0] TxD_state = 0;
wire TxD_ready = (TxD_state==0);
assign TxD_busy = ~TxD_ready;

reg [7:0] TxD_shift = 0;
always @(posedge clk)
begin
    if(TxD_ready & TxD_start)
        TxD_shift <= TxD_data;
    else
    if((TxD_state>4'b0001) & TxDTick)
        TxD_shift <= (TxD_shift >> 1);

    case(TxD_state)
        4'b0000: if(TxD_start) TxD_state <= 4'b0001;
        4'b0001: if(TxDTick) TxD_state <= 4'b0010;  // start bit
        4'b0010: if(TxDTick) TxD_state <= 4'b0011;  // bit 0
        4'b0011: if(TxDTick) TxD_state <= 4'b0100;  // bit 1
        4'b0100: if(TxDTick) TxD_state <= 4'b0101;  // bit 2
        4'b0101: if(TxDTick) TxD_state <= 4'b0110;  // bit 3
        4'b0110: if(TxDTick) TxD_state <= 4'b0111;  // bit 4
        4'b0111: if(TxDTick) TxD_state <= 4'b1000;  // bit 5
        4'b1000: if(TxDTick) TxD_state <= 4'b1001;  // bit 6
        4'b1001: if(TxDTick) TxD_state <= 4'b1010;  // bit 7
        4'b1010: if(TxDTick) TxD_state <= 4'b0000;  // stop1
        default: if(TxDTick) TxD_state <= 4'b0000;
    endcase
end
```

- تعیین خروجی فرستنده

طبق قراردادی که بین گیرنده و فرستنده هست، وقتی فرستنده در حال ارسال داده نیست، بیت یک روی باس قرار دارد پس خروجی فرستنده را با این شرط که در استیتی ارسال داده نیستیم or میکنیم (اگر در حال ارسال داده نباشیم یک روی باس قرار میگیرد. در غیر این صورت هم (که با شرط txd_state>0001 مشخص شده است) بیت صفر شیفت رجیستر را ارسال میکنیم (که در استیت بعد دور ریخته میشود).

```
assign TxD = (TxD_state == 4'b0000) | ((TxD_state > 4'b0001) & TxD_shift[0]) | (TxD_state == 4'b1010);
endmodule
```

- نتیجه شبیه سازی transmitter:



Async_receiver

```
module async_receiver(
    input clk,
    input RxD,
    output reg RxD_data_ready = 0,
    output reg [7:0] RxD_data = 0
);
```

- مقادیر پارامترها

```
parameter ClkFrequency = 50000000;
parameter Baud = 115200;
parameter Oversampling = 4; // needs to be a power of 2
```

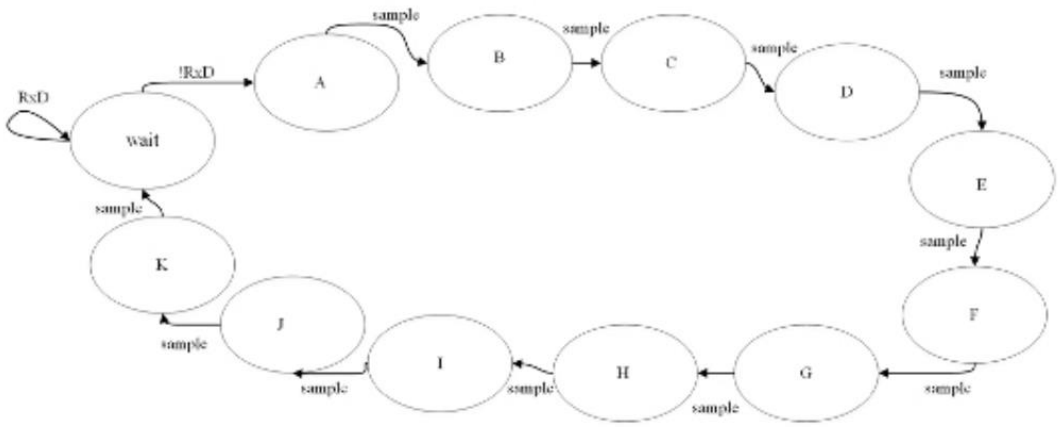
- گرفتن اینستنس از ماژول baud rate generator

زمانی که در حال دریافت داده باشد BaudTickGen فعال میشود (چون در تمام stateها غیر از 0000 ریسیور در حال دریافت داده است و enable که active low است به این شرط حساس است).

خروجی این ماژول هم به sample وصل شده است.

```
BaudTickGen #(ClkFrequency, Baud, Oversampling) tickgen(.clk(clk), .enable((RxD_state==4'b0000)), .tick(sample));
//assign RxD_data_ready = (RxD_state == 4'b1011);
```

- state machine



همانطور که در شکل دیده میشود 12 استیت داریم که در بدنه always زیر پیاده سازی شده است. بعد از دریافت start bit بار اول به اندازه 2 بار tick زدن (که با count میشماریم) صبر میکنیم تا در tick دوم sample بگیریم. بعد از این اگر هر بار به اندازه 4 تا tick صبر کنیم (که با cout مشخص میشود و در قسمت بعد، پیاده سازی آن آمده است)، هر بار در tick دوم sample گرفته ایم.

ضمنا اگر در حالت wait نباشیم، در هر state هشت بیتِ دیتای خروجی را با یک بیت ورودی، شیفت میدهیم که در بدنه else if آمده است.

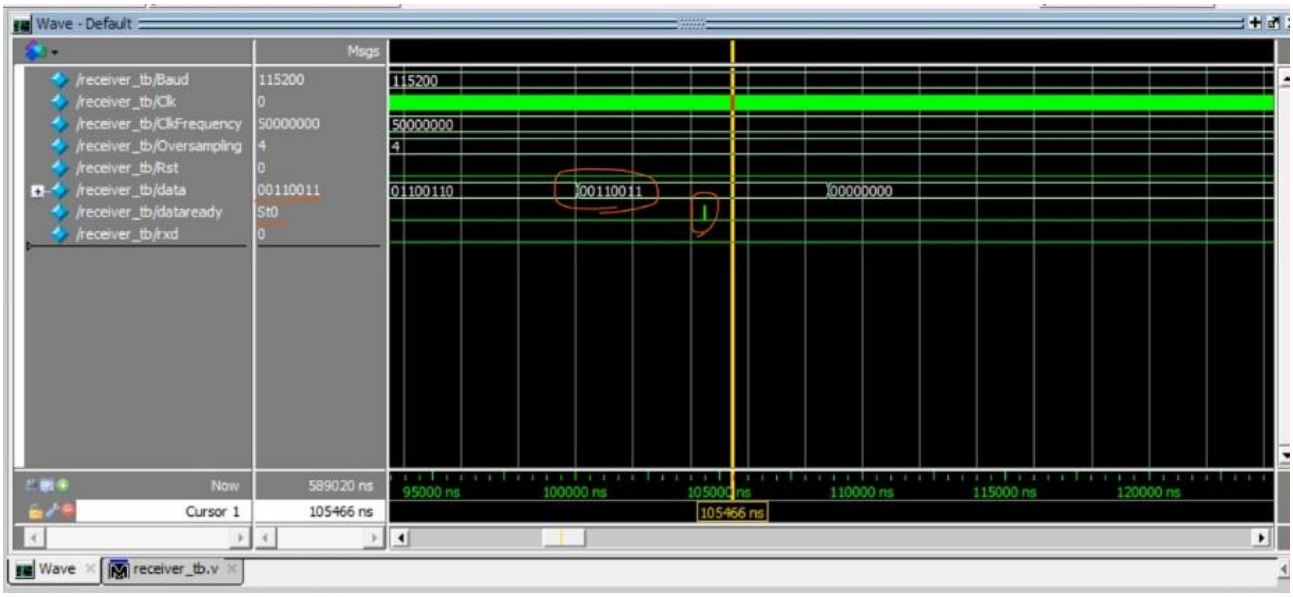
```
always @(posedge clk)
begin

    if((RxD_state == 4'b0000) & !RxD)
        RxD_data[0]= RxD;

    else if((RxD_state > 4'b0010) & cout)
        RxD_data <= { RxD, RxD_data[7:1]};

     case(RxD_state)
        4'b0000: if(!RxD) RxD_state <= 4'b0001;
        4'b0001: if(count == 2'b10) RxD_state <= 4'b0010;  // start bit
        4'b0010: if(cout) RxD_state <= 4'b0011;  // start bit sampling
        4'b0011: if(cout) RxD_state <= 4'b0100;  // bit 0
        4'b0100: if(cout) RxD_state <= 4'b0101;  // bit 1
        4'b0101: if(cout) RxD_state <= 4'b0110;  // bit 2
        4'b0110: if(cout) RxD_state <= 4'b0111;  // bit 3
        4'b0111: if(cout) RxD_state <= 4'b1000;  // bit 4
        4'b1000: if(cout) RxD_state <= 4'b1001;  // bit 5
        4'b1001: if(cout) RxD_state <= 4'b1010;  // bit 6
        4'b1010: begin if(cout) RxD_state <= 4'b1011; end // bit 7
        4'b1011: begin if(sample) RxD_state <= 4'b0000;  RxD_data_ready=1; end  // stop1
        default: if(RxD) RxD_state <= 4'b0000;
     endcase
end
```

- تعیین count و cout

همانطور که در قسمت قبل توضیح داده شد بار اول نیاز است به اندازه دو تیک (که خروجی تیک به sample وصل شده است) صبر کنیم و با count یک شدن sample را میشماریم. پس از این هم باید به اندازه 4 تیک صبر کنیم، لذا با هر بار که count از صفر به سه میرسد، cout را یک میکنیم و همانطور که در قسمت قبل داده شد، غیر از دو استیت ابتدایی، باقی استیت ها با cout تغییر میکنند.

```
always@( sample)begin

      if (RxD_state == 4'b0000) begin
    count = 2'b00;
    cout = 0; end
     else if ( sample)begin

    count = count + 1;
    cout = 0; end

     if (count == 2'b11 & sample)
      cout = 1'b1;
      else if(count == 2'b11 & ~sample)
     cout = 1'b0;
end


endmodule
```

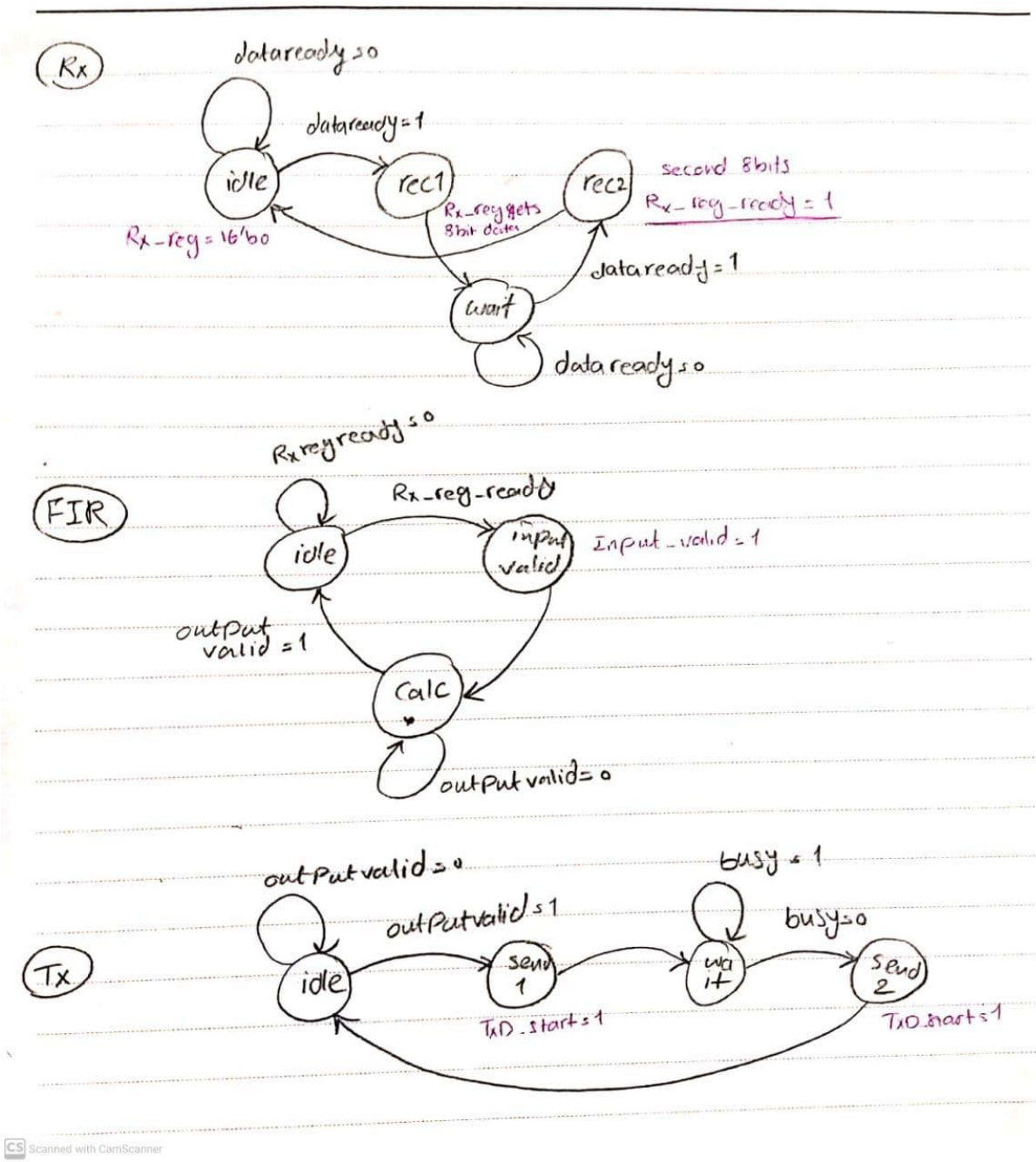
- نتیجه شبیه سازی receiver:



راه اندازی و تست سیستم کلی

واحد کنترل

پیاده سازی با ماشین حالت :



واحد کنترل این پروژه برای اینکه بتواند حالت دوطرفه بودن انتقالات را رعایت کند شامل سه بخش جداگانه برای هر یک از قسمت های دریافت کننده و ارسال کننده و فیلتر است.

کنترلر دریافت کننده

```
//RX
always@(presentState_RX, data_ready)
begin
    //nextState <= start;// we must initialize out in order to synthesize tool don't make a latch.
    case(presentState_RX)
    idel_RX: if(data_ready == 1'b1) nextState_RX = recive_1_RX; else nextState_RX = idel_RX;
    recive_1_RX: nextState_RX = wait_1_RX;
    wait_1_RX: if(data_ready == 1'b1) nextState_RX = recive_2_RX; else nextState_RX = wait_1_RX;
    recive_2_RX: nextState_RX = idel_RX;
    endcase
end
```

توضیح عملکرد:

یک رجیستر 16 بیتی برای دریافت دو مرحله ای داده در نظر میگیریم.

ابتدا در استیت idel_RX صبر می کند تا زمانی که سیگنال data_ready صادر شود. این سیگنال به این معنی است که دیتا آماده دریافت است. پس به استیت recive_1_RX می رود که در این استیت فرمان شیفت دادن رجیستر به اندازه 8 بیت و درنتیجه دریافت 8 بیت اول داده صادر می شود.

سپس دوباره به استیتی می رود که منتظر سیگنال data_ready بماند و هر وقت این سیگنال یک شد به استیت بعدی رفته و 8 بیت دیگر داده دریافت میکند و بدین ترتیب رجیستر 16 بیتی کاملا پر می شود که در اینجا سیگنال RX_reg_ready را برای اعلام اینکه همه داده ها دریافت شده منتشر می کند.

کنترلر ارسال کننده

```
//TX
always@(presentState_TX, Output_Valid_FIR, busy)
begin
    //nextState <= start;// we must initialize out in order to synthesize tool don't make a latch.
    case(presentState_TX)
    idel_TX: if(Output_Valid_FIR == 1'b1) nextState_TX = send_1_TX; else nextState_TX = idel_TX;
    send_1_TX: nextState_TX = wait_1_TX;
    wait_1_TX: if(busy == 1'b0) nextState_TX = send_2_TX; else nextState_TX = wait_1_TX;
    send_2_TX: nextState_TX = idel_TX;
    endcase
end
```

توضیح عملکرد:

در کنترلر ارسال کننده، منتظر سیگنال Output_Valid_FIR می مانیم تا مطمئن شویم دیتایی که مدنظر است تولید شده و آماده ارسال است.

سپس به استیت send_1_TX می رود که 8 بیت اول داده را در آن ارسال می کند. به این صورت که رجیستری که 16 بیتی است و خروجی فیلتر را ذخیره می کند شیفت می دهد که 8 بیت اول از این رجیستر خارج شود. سپس به استیت wait_1_TX می رود که تا زمانی که سیگنال busy یک باشد و ارسال کننده مشغول باشد در آنجا می ماند.

به محض اینکه کار این قسمت تمام شود به سراغ ارسال قسمت دوم داده (8 بیت دوم) به همان صورت قبل می رود و بعد به استیت idel_TX برمی گردد.

کنترلر فیلتر

```
//FIR
always@(presentState_FIR, RX_reg_ready, Output_Valid_FIR)
begin
    //nextState <= start;// we must initialize out in order to synthesize tool don't make a latch.
    case(presentState_FIR)
    idel_FIR: if(RX_reg_ready == 1'b1) nextState_FIR = inputValidation_FIR; else nextState_FIR = idel_FIR;
    inputValidation_FIR: nextState_FIR = calculation_FIR;
    calculation_FIR: if(Output_Valid_FIR == 1'b1) nextState_FIR = idel_FIR; else nextState_FIR = calculation_FIR;
    endcase
end
```

توضیح عملکرد:

این کنترلر تا زمانی که مطمئن شویم 16 بیت داده توسط دریافت کننده دریافت شده و در رجیستر آماده است در استیت اول می ماند. سپس سیگنال معتبر بودن ورودی را صادر می کند تا فیلتر راه اندازی شود. و تا زمانی که فیلتر با سیگنال معتبر بودن خروجی، اتمام محاسباتش را اعلام نکرده در این حالت می ماند.

سپس مجدد به استیت اولیه بر میگردد و منتظر می ماند.

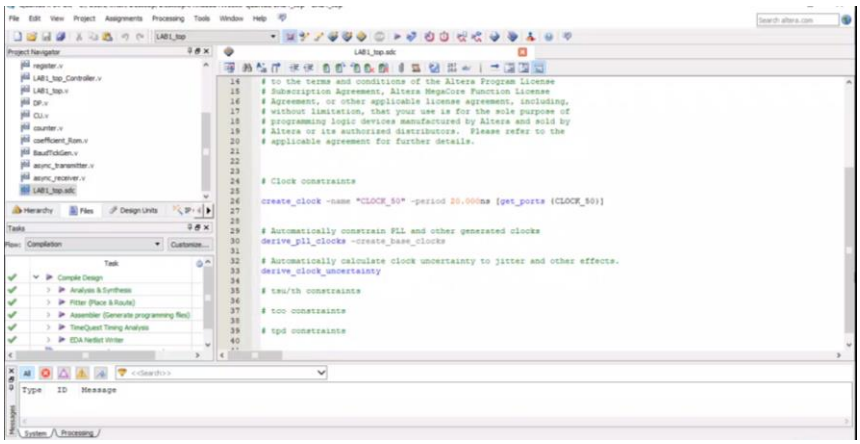
توضیح عملکرد کد متلب

در ابتدا یک پورت سریال (روی COM3) ایجاد میکنیم و تنظیمات پورت را انجام میدهیم. سپس فایل صوتی را لود کرده و با تایپ دابل به عنوان input ذخیره میکنیم و output به اندازه سایز input و با مقدار صفر میسازیم. سپس شروع به فرستادن (روی پورت fwrite میکنیم) دیتا (inputs) میکنیم. چون گیرنده و فرستنده 8 بیتی اند به این شکل ارسال میکنیم (خارج قسمت و باقی مانده شان به 256). سپس دیتا را از روی پورت دریافت (fread) و ذخیره میکنیم. با توجه به اینکه برای ارسال، ورودی‌ها را تقسیم بر 256 کرده و خارج قسمت و باقی مانده را ارسال کرده بودیم، برای بازسازی، دیتاهای فرد را ضربدر 256 و به اضافه دیتای بعدی میکنیم و به عنوان خروجی ذخیره میکنیم. سپس دیتاهای خروجی fpga را به داده‌هایی که قابل تبدیل به فایل صوتی اند تبدیل کرده و خروجی نهایی (فایل صوتی) را با دیتاهای فیلتر شده میسازیم.

صوت خروجی در فایل گزارش آمده است.

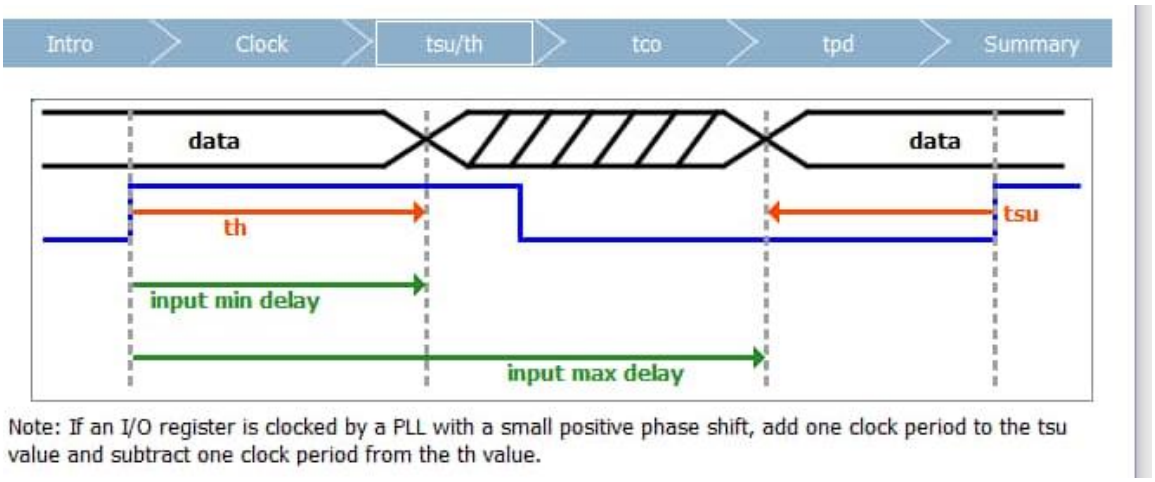
اضافه کردن فایل Analyzer Timing analyzer wizard:

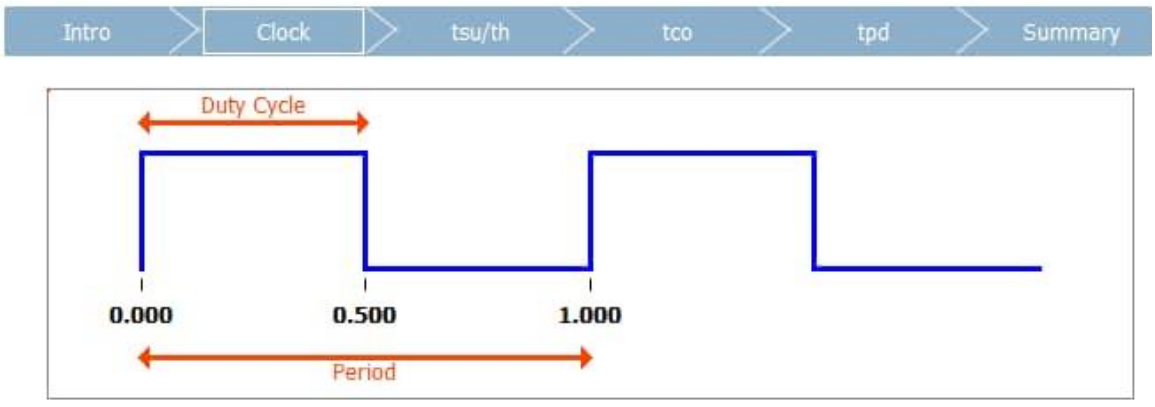
فایل sdc. تولید شده و تعیین کلاک 50MHz در شکل زیر آمده است.

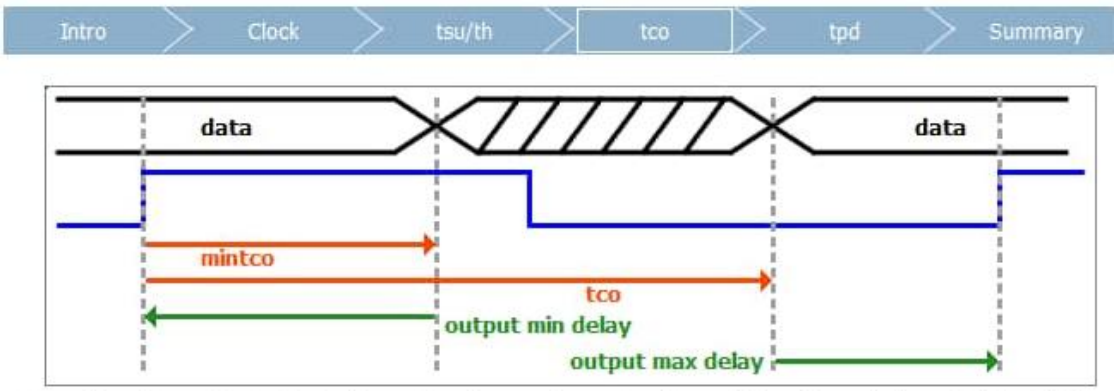


ابزار Analyzer Timing analyzer wizard یک static timing analyzer برای FPGA فراهم میکند. فایل sdc مهمترین timing requirement های دیزاین را تعریف میکند و وقتی که با یک PLL کلاک generate میشود، فایل sdc کلاک‌های base, external, virtual میسازد و همچنین تایمینگ دیوایس‌های IO را هم بر اساس virtual clk تعریف میکند و این فایل timing exception ها را هم در خود دارد.

- محدودیت‌هایی که می‌توان اضافه کرد:



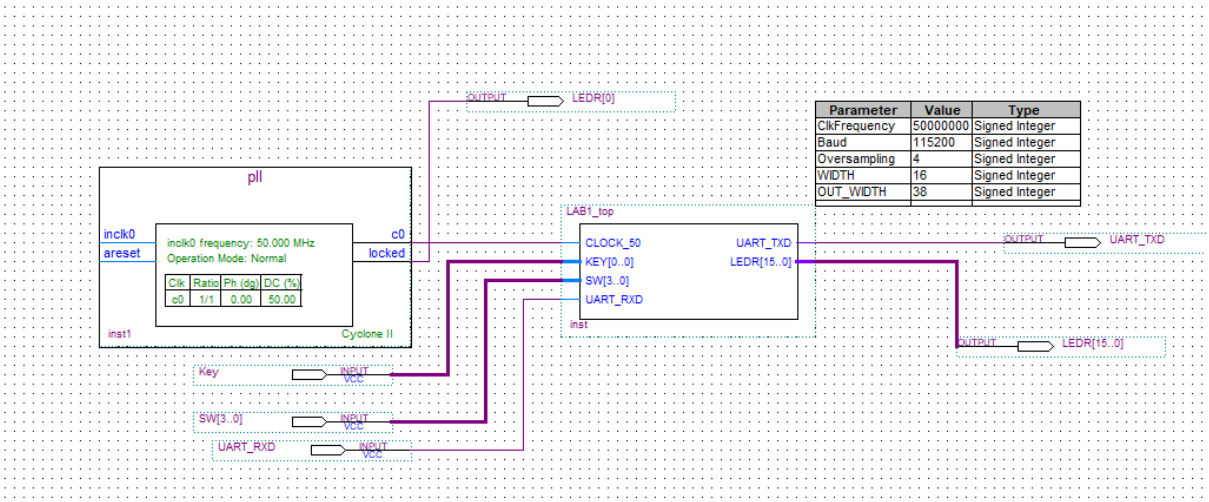




Note: If an I/O register is clocked by a PLL with a small negative phase shift, add one clock period to both the tco and minimum tco values.

The TimeQuest Timing Analyzer Wizard allows you to create initial timing constraints for your design, including clock settings, tSU, tH, and tPD constraints. If your design uses a phase-locked loop (PLL), you can use the wizard to specify the input frequency of the pin and to create the settings to generate the PLL output clocks.

اضافه کردن یک PLL به صورت یک ماژول و اتصال اون به کلاک سیستم:



نحوه ی اساین کردن کلاک به یک گلوبال کلاک: