به نام خداوند بخشنده مهربان

طراحی سیستم های دیجیتال

دکتر اجلالی

۴۰۱۱۰۶۳۲۸ - نیکا قادری
بهار ۱۴۰۳

تاریخ گزارش: ۱ خرداد ۱۴۰۳ | **طراحی واحد** $uart$ | سوال نهم

فهرست مطالب

۱ معرفی و بررسی هدف تمرین

هدف از انجام این آزمایش طراحی یک $Universal\,Asynchronous\,Receiver\,Transmitter - UART$ است. $UART$ یکی از پر استفاده ترین پروتکل های ارتباط $device - to - device$ است که از دو بخش تشکیل شده: یک بخش فرستنده و یک بخش گیرنده. هدف این است که اگر یک داده ی موازی داریم، بتوانیم آن را به صورت سری روی یک تک سیم منتقل کنیم. یعنی هدف اصلی این تکنولوژی تبدیل داده موازی به داده سریال است. و یا اینکه این داده سریال $Decode$ شود و آن را به داده موازی برگرداند. برای مثال در بخش فرستنده ۱۰ بیت ارسال می شود که بیت اول نشان دهنده شروع[۱]، بیت دوم توازن[۲]، ۷ بیت بعدی داده ارسال شده[۳] و بیت آخر پایان[۴] می باشد.

۲ ماژول ها و توضیحات

۱.۲ sender

ورودی ها

۱. $rstN$: ورودی ریست، با صفر شدن آن کل خروجی ها و حالت های مدار صفر می شود.

۲. clk: سیگنال ساعت

۳. $start$: سیگنالی که مشخص می کند واحد باید شروع به ارسال اطلاعات کند.

۴. $data_in$: یک بردار[۵] که شامل هفت بیت می باشد. داده ای که می خواهیم ارسال شود را در این ورودی قرار می دهیم.

خروجی ها

۱. s_out: بیت خروجی می باشد. چون داده را به صورت سری ارسال می کنیم، این بیت در هر کلاک یکی از بیت های داده ورودی را مشخص می کند و می فرستد.

۲. $sent$: بیتی که مشخص می کند ارسال داده ها تمام شده است یا خیر.

مقادیر تعریف شده در بدنه درونی کد

۱. $state$: رجیستری دو بیتی که حالت کنونی مدار را نگه می دارد.

---

[۱] $start$
[۲] $parity$
[۳] $senddata$
[۴] $finish$
[۵] vector

۲. $data$: برداری هفت بیتی که با داده ورودی برابر است. علت تعریف این رجیستر جلوگیری از ناسازگاری در مدار است، یعنی اگر مثلا در حین انجام عملیات ورودی تغییر کرد، این حالت غیر مجاز در نظر گرفته شود و مقدار این رجیستر تغییری نکند تا مدار بتواند به کار خود ادامه دهد.

۳. $data_index$: بیتی که در حال حاضر در حال فرستاده شدن است را مشخص می کند و مقادیر آن از بین صفر تا شش متغیر است.

۴. $parity\_sig$: برابر با xor بیت های $data$ می باشد. این داده با تایپ $wire$ تعریف شده است.

۵. S_IDLE، S_START، S_PARITY، S_SEND، S_STOP: پارمتر های محلی که حالت های مختلف ماژول را نمایش می‌دهند و به ترتیب با مقادیری ۰ تا ۴ مقدار دهی شده اند. کاربرد این پارامتر ها خوانایی بیشتر کد می باشد، تا هنگامی که می خواهیم $state$ را مقداردهی کنیم به جای اعداد، از این پارامترها استفاده کنیم.

```
module sender # (
    parameter START_SIG = 1
) (
    input           rstN,
    input           clk,
    input           start,
    input   [6:0]   data_in,
    output  reg     s_out,
    output  reg     sent
);

localparam S_IDLE       = 0;
localparam S_START      = 1;
localparam S_PARITY     = 2;
localparam S_SEND       = 3;
localparam S_STOP       = 4;

reg [2:0]   state;
reg [6:0]   data;
reg [2:0]   data_index;

wire parity_sig;
assign parity_sig = ^data;
```

شکل ۱: ورودی و خروجی ها

حال بدنه اصلی این ماژول را شرح می دهیم . بررسی و مدیریت تمام حالت ها در یک بلاک $always$ انجام می شود.

```
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        state <= S_IDLE;
        data_index <= 0;
        s_out <= 0;
        sent <= 0;
    end
    else begin
        case (state)
            S_IDLE: begin
                if (start) begin
                    data_index <= 0;
                    data <= data_in;
                    state <= S_START;
                    sent <= 0;
                end
            end
            S_START: begin
                s_out <= START_SIG;
                state <= S_PARITY;
            end
            S_PARITY: begin
                s_out <= parity_sig;
                state <= S_SEND;
            end
            S_SEND: begin
                s_out <= data[data_index];
                data_index <= data_index + 1;
                if (data_index == 6)
                    state <= S_STOP;
            end
            S_STOP: begin
                s_out <= ~START_SIG;
                state <= S_IDLE;
                sent <= 1;
            end
            default: state <= S_IDLE;
        endcase
    end
end
```

شکل ۲: ماژول $sender$

در شکل ۲ ، این بلاک پیاده سازی شده است. در ابتدا، اگر سیگنال ریست فعال باشد باید $state$، ایندکس داده، و همچنین تمام خروجی ها برابر با صفر شود.

در غیر این صورت، باید بر اساس حالت کنونی تصمیم بگیریم.
اگر حالت کنونی:

- S_IDLE باشد:
اگر $start$ فعال نباشد باید در حالت فعلی بمانیم و وضعیت ماژول هیچ تغییری نمی کند.
اگر $start$ فعال باشد، اندیس دیتا صفر می شود. $data$ را از ورودی $data\_in$ می گیریم و ذخیره می کنیم.
سیگنال $send$ صفر می شود.
حالت بعدی برابر با S_START می شود.

- S_START باشد:
مقدار ۱ می فرستیم بدین معنی که برنامه شروع به کار کرده است.
همچنین باید به حالت بعدی برویم که برابر با S_PARITY می باشد.

- S_PARITY باشد:
خروجی s_out را برابر با بیت توازن می گذاریم تا فرستاده شود.
همچنین حالت بعدی برابر با S_SEND می شود.

- S_SEND باشد:
در این حالت باید بیت های موجود در $data$ را در s_out قرار دهیم تا یکی یکی فرستاده شوند.
همچنین باید در مرحله اندیس داده را یک واحد افزایش دهیم؛ و اگر اندیس برابر با ۶ شد، یعنی تمام داده فرستاده شد، باید به حالت بعدی - که S_STOP است - برویم. (اگر کمتر از شش بود همچنان در این حالت می مانیم.)

- S_STOP باشد:
s_out صفر می شود.
$sent$ یک می شود که نشان دهنده این می باشد که ارسال تکمیل شده است.
دوباره به حالت اول که S_IDLE می باشد می رویم.

در حالت پیش فرض نیز حالت اولیه برابر با S_IDLE فرض می شود. بدین صورت ماژول $sender$ تکمیل می شود.

۲.۲ reciever

ورودی ها

۱. $rstN$: ورودی ریست، با صفر شدن آن کل خروجی ها و حالت های مدار صفر می شود.

۲. clk: سیگنال ساعت

۳. s_in: تک بیتی است که فرستنده در هر کلاک تحت عنوان s_out برای گیرنده می فرستد. توجه کنید که انتقال اطلاعات به صورت سری است و در هر کلاک یک بیت برای گیرنده فرستاده می شود.

خروجی ها

۱. $received$: مشخص می کند داده به طور کامل دریافت شده است یا نه.

۲. $check\_parity$: با مقایسه بیت توازنی که در داده های ورودی وارد می شود و بیت توازن محاسبه شده توسط خود برنامه مشخص می شود داده به درستی دریافت شده است یا خیر.

۳. $data$: یک رجیستر هفت بیتی که داده ای که ماژول فرستنده می فرستد را ذخیره می کند. در واقع داده سری را به صورت موازی بازسازی می کند.

مقادیر تعریف شده در بدنه درونی کد

۱. $state$: رجیستری دو بیتی که حالت کنونی مدار را نگه می دارد.

۲. $data$: برداری هفت بیتی که با داده ورودی برابر است. علت تعریف این رجیستر جلوگیری از ناسازگاری در مدار است، یعنی اگر مثلا در حین انجام عملیات ورودی تغییر کرد، این حالت غیر مجاز در نظر گرفته شود و مقدار این رجیستر تغییری نکند تا مدار بتواند به کار خود ادامه دهد.

۳. $data\_index$: اندیس کنونی دیتای دریافت شده را مشخص می کند و بین صفر تا شش متغیر است.

۴. $expected\_parity$: بیت توازنی که ورودی گرفته می شود.

۵. $actual\_parity$: بیت توازنی که در برنامه و بر اساس ورودی داده شده مشخص می شود.

۶. S_IDLE، S_PARITY، $S\_RECEIVE$، S_STOP: پارمتر های محلی که حالت های مختلف ماژول را نمایش می دهند و به ترتیب با مقادری ۰ تا ۳ مقدار دهی شده اند. کاربرد این پارامتر ها خوانایی بیشتر کد می باشد، تا هنگامی که می خواهیم $state$ را مقداردهی کنیم به جای اعداد، از این پارامترها استفاده کنیم.

```
module receiver # (
    parameter START_SIG = 1
) (
    input               rstN,
    input               clk,
    input               s_in,
    output  reg         received,
    output              check_parity,
    output  reg [6:0]   data
);

localparam S_IDLE       = 0;
localparam S_PARITY     = 1;
localparam S_RECEIVE    = 2;
localparam S_STOP       = 3;

reg [1:0]   state;
reg [2:0]   data_index;
reg         expected_parity;
wire        actual_parity;

assign actual_parity = ^data;
assign check_parity = actual_parity == expected_parity;
```

شکل ۳: ورودی ها و خروجی ها

حال بدنه اصلی کد را شرح می دهیم که در یک بلاک $always$ تعریف می شود.

```
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        state <= S_IDLE;
        data_index <= 0;
        received <= 0;
        data <= 0;
    end
    else begin
        case (state)
            S_IDLE: begin
                if (s_in == START_SIG) begin
                    data_index <= 0;
                    data <= 0;
                    state <= S_PARITY;
                    received <= 0;
                end
            end
            S_PARITY: begin
                expected_parity <= s_in;
                state <= S_RECEIVE;
            end
            S_RECEIVE: begin
                data[data_index] <= s_in;
                data_index <= data_index + 1;
                if (data_index == 6) begin
                    state <= S_STOP;
                end
            end
            S_STOP: begin
                state <= S_IDLE;
                received <= 1;
            end
            default: state <= S_IDLE;
        endcase
    end
end

endmodule
```

شکل ۴: ماژول $reciever$

در ابتدا، اگر سیگنال ریست فعال باشد باید $state$، ایندکس داده، و همچنین تمام خروجی ها برابر با صفر شود.
در غیر این صورت، باید بر اساس حالت کنونی تصمیم بگیریم.
اگر حالت کنونی:

- S_IDLE باشد:
  اگر s_in فعال نباشد باید در حالت فعلی بمانیم و وضعیت ماژول هیچ تغییری نمی کند.
  اگر s_in فعال باشد، اندیس دیتا صفر می شود.
  $data$ و $recieved$ نیز صفر می شوند.
  حالت بعدی برابر با S_PARITY می شود.
- S_PARITY باشد:
  خروجی s_in را برابر با بیت توازن ورودی می گذاریم تا دریافت شود، چون دومین بیت ارسالی بیت توازن است.
  همچنین حالت بعدی برابر با $S\_RECEIVE$ می شود.
- $S\_RECEIVE$ باشد:
  در این حالت باید بیت ورودی را در اندیس مناسب از $data$ لود کنیم.
  بعد از لود داده، اندیس یک واحد زیاد می شود و اگر برابر با شش شد، بدین معنی است که داده به طور کامل دریافت شده است. در این حالت باید به حالت بعدی، یعنی S_STOP برویم.
- S_STOP باشد:
  $received$ یک می شود که نشان دهنده این می باشد که دریافت تکمیل شده است.
  دوباره به حالت اول که S_IDLE می باشد می رویم.

در حالت پیش فرض نیز حالت اولیه برابر با S_IDLE فرض می شود. بدین صورت ماژول $reciever$ تکمیل می شود.

uart ۳.۲

ورودی ها

۱. $rstN$: ورودی ریست، با صفر شدن آن کل خروجی ها و حالت های مدار صفر می شود.

۲. clk: سیگنال ساعت

۳. $clk۲$: سیگنال ساعت دوم؛ از آنجا که در صورت مساله خواسته شده سیگنال های ساعت فرستنده و گیرنده با هم متفاوت باشند.

۴. s_in: ورودی s_in در گیرنده. تک بیتی از داده اصلی که در هر کلاک دریافت می شود.

۵. $send$: ورودی $send$ در فرستنده. تک بیتی از داده اصلی که در هر کلاک فرستاده می شود.

۶. $send_data$: داده ای که قرار است به واحد دیگری فرستاده شود

خروجی ها

۱. s_out: تک بیتی که با آن داده را مرحله به مرحله به یک واحد دیگر می فرستیم.

۲. $sent$: سیگنالی که فرستنده می فرستد و به معنای اتمام فرآیند ارسال است.

۳. $received$: سیگنالی که گیرنده می فرستد و به دلیل تکمیل فرآیند دریافت اطلاعات است.

۴. $received\_data$: داده ی دریافت شده ی نهایی توسط گیرنده.

۵. $check\_receive\_parity$: سیگنالی که گیرنده می فرستد و مشخص می کند آیا داده به طور کامل و بدون اشتباه دریافت شده است یا خیر.

شکل ۵ این ماژول را نشان می دهد. کافی است در هر واحد $uart$، یک بار از فرستنده و بار دیگر از گیرنده نمونه[۶] بگیریم و ورودی ها و خروجی های متناظر را به هم متصل کنیم. توجه کنید که یک پارامتر $START\_SIG$ هم داریم که به طور پیش فرض با یک مقدار دهی شده است و منطق شروع و پایان انتقال داده را تعیین می کند. یعنی بیت شروع یک باشد و بیت پایان صفر، یا برعکس.

```
module uart #(
    parameter START_SIG = 1
) (
    input           rstN,
    input           clk,
    input           clk2,
    input           s_in,
    input           send,
    input   [6:0]   send_data,
    output          s_out,
    output          sent,
    output          received,
    output  [6:0]   received_data,
    output          check_receive_parity
);

sender #(START_SIG) SENDER (rstN, clk, send, send_data, s_out, sent);
receiver #(START_SIG) RECEIVER (rstN, clk2, s_in, received, check_receive_parity, received_data);

endmodule
```

شکل ۵: واحد $uart$

testbench ۴.۲

در ابتدا مقادیر مورد نیاز را مانند بخش های قبل، و طبق تصویر ۶ تعریف و در صورت نیاز، مقداردهی اولیه می کنیم.
همچنین دو سیگنال کلاک متفاوت تولید می کنیم و در ابتدای برنامه سیگنال ریست را فعال می کنیم.

[۶] instance

```
module testbench ();
localparam START_SIG = 1;

reg     rstN = 0, clk = 1, clk2 = 1;
reg     send_1, send_2;
reg     [6:0]   send_data_1, send_data_2;

wire    sent_1, sent_2;
wire    received_1, received_2;
wire    [6:0]   received_data_1, received_data_2;
wire    check_1, check_2;

uart #(START_SIG) U1 (rstN, clk, clk2, s_1, send_1, send_data_1, s_2, sent_1, received_1, received_data_1, check_1);
uart #(START_SIG) U2 (rstN, clk2, clk, s_2, send_2, send_data_2, s_1, sent_2, received_2, received_data_2, check_2);

always #10 clk = ~clk;
always #5 clk2 = ~clk2;

initial begin
    #40 rstN = 1;
end
```

شکل ۶: مقداردهی های اولیه

سپس، دو داده مختلف را بین این دو واحد منتقل می کنیم. در مرحله ی اول، با توجه به شکل ۷ ابتدا داده مورد نظر را - که رشته ای به از کاراکتر های هفت بیتی با مقدار $'Eagle'$ می باشد - در ۱_$string$ ذخیره می کنیم. سپس، آن را به عنوان ورودی فرستنده واحد اول قرار می دهیم و سیگنال ۱_$send$ در آن را - که فرمان شروع عملیات ارسال را می دهد - برای مدتی فعال می کنیم.
سپس، منتظر می مانیم تا واحد اول، تکمیل ارسال را اعلام کند و سپس واحد دوم نیز، اتمام دریافت داده را اعلام کرده و داده ی دریافت شده را نشان دهد.

```
wire [6:0]  string_1 [4:0];
assign string_1[4] = "E";
assign string_1[3] = "a";
assign string_1[2] = "g";
assign string_1[1] = "l";
assign string_1[0] = "e";
integer i;

initial begin
    #50;
    for (i = 5; i > 0; i = i - 1) begin
        send_data_1 = string_1[i-1];
        #10 send_1 = 1;
        #20 send_1 = 0;

        wait (sent_1);
        $display("%c sent from U1", send_data_1);
        wait (received_2);
        $display("%c received by U2. check: %d", received_data_2, check_2);
    end
    $stop();
end
```

شکل ۷: مرحله اول

در مرحله بعد، همین کار را دوباره برای واحد های یک و دو انجام می دهیم، با این تفاوت که این بار واحد دو اطلاعات را می فرستد و واحد یک دریافت می کند. با توجه به شکل ۸ ، دوباره یک رشته از کاراکتر های هفت بیتی - که این بار کلمه $'Wolf'$ را نشان می دهند، تعریف می کنیم و بعد در یک حلقه، هر بار یکی از خانه های آن را در ۲_$string$ لود کرده، و سیگنال ارسال واحد دوم را فعال می کنیم. سپس مانند مرحله قبلی سیگنال های نتیجه را در ترمینال نمایش می دهیم.

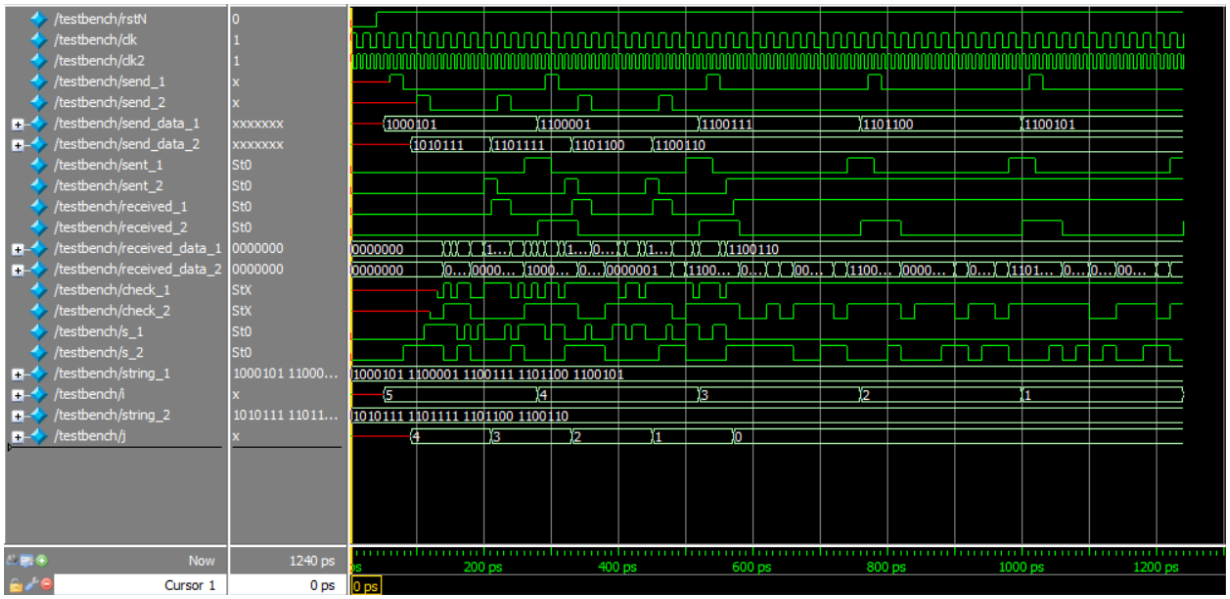
```
wire [6:0]  string_2 [3:0];
assign string_2[3] = "W";
assign string_2[2] = "o";
assign string_2[1] = "l";
assign string_2[0] = "f";
integer j;

initial begin
    #90;
    for (j = 4; j > 0; j = j - 1) begin
        send_data_2 = string_2[j-1];
        #10 send_2 = 1;
        #20 send_2 = 0;

        wait (sent_2);
        $display("%c sent from U2", send_data_2);
        wait (received_1);
        $display("%c received by U1. check: %d", received_data_1, check_1);
    end
end

endmodule
```

شکل ۸: مرحله دوم

بدین ترتیب، تست بنچ نیز کامل می شود.

۳ اجرا و شکل موج

نتیجه ی اجرای تست بنچ به صورت زیر می باشد:



شکل ۹: شکل موج

همچنین خروجی ترمینال به صورت زیر می باشد:

```
VSIM 7> run -all
# W sent from U2
# W received by U1. check: 1
# E sent from U1
# E received by U2. check: 1
# o sent from U2
# o received by U1. check: 1
# l sent from U2
# l received by U1. check: 1
# a sent from U1
# a received by U2. check: 1
# f sent from U2
# f received by U1. check: 1
# g sent from U1
# g received by U2. check: 1
# l sent from U1
# l received by U2. check: 1
# e sent from U1
# e received by U2. check: 1
```

شکل ۱۰: خروجی ترمینال

همان طور که مشاهده می شود انتقال اطلاعات به درستی، و به صورت موازی انجام می شود، بدین معنی که واحد فرستنده و گیرنده در یک واحد $uart$، هیچ ارتباطی با هم ندارند.

۴ نتیجه گیری

در این سوال، عملکرد واحد های $uart$ و اهمیت آن در ارتباط بین دو واحد مجزا که به طور فیزیکی به هم متصل شده اند مورد بررسی قرار گرفت و یک واحد از آن، با استفاده از ماژول های فرستنده و گیرنده ساخته شد. از آنجایی که فرض شده این دو ماژول در یک واحد $uart$ ارتباطی به هم ندارند، سیگنال های ساعت آنها به طور مجزا طراحی شده است.
سپس تست بنچی برای بررسی عملکرد این واحد ساختیم و در آن، خروجی های فرستنده واحد اول را به ورودی های گیرنده واحد دوم، و خروجی های فرستنده واحد دوم را به ورودی های گیرنده واحد دوم متصل کردیم. این روش باعث می شود بتوان در یک واحد $uart$ همزمان داده ای را ارسال کرد و داده دیگری را دریافت کرد. در آخر تست بنچ را اجرا کردیم تا از صحت برنامه نوشته شده اطمینان حاصل شود.

پایان