

گزارش کار آزمایشگاه DSD

**آزمایش شماره 7**



11 اردیبهشت 1400

عرشیا اخوان

محمدحسین عبدی

علیرضا ایلامی

|  |  |  |
| --- | --- | --- |
| تاریخ آزمایش: 11 اردیبهشت 1400 | موضوع: UART | **شماره آزمایش: 7** |
| **علیرضا ایلامی**  **97101286** | **محمدحسین عبدی**  **97110285** | **عرشیا اخوان**  **97110422** |

مقدمه:

هدف از انجام این آزمایش طراحی یک Universal Asynchronous Receiver Transmitter - UART است.

شرح آزمایش:

یو آرت (UART) چیست؟

UART یکی از پر استفاده ترین پروتکل های ارتباط device-to-device است.

UART از دو بخش تشکیل شده است. یک بخش فرستنده و یک بخش گیرنده.

هدف این است که اگر یک داده ی موازی داریم، بتوانیم این را به صورت سری روی یک تک سیم منتقل کنیم. یعنی کلا هدفش تبدیل داده موازی به داده سریال است. و اینکه این داده سریال Decode شود و آن را به داده موازی برگرداند.

در واقع هدف اصلی UART انتقال داده در مسیرهای طولانی و اطمینان از سلامت داده دریافتی (به کمک بیت Parity که بعدا به آن می پردازیم) است.

بخش فرستنده:

یک داده موازی (در این آزمایش به پهنای 7 بیت) داریم.

این ها را باید به بیت های سری تبدیل کنیم.

ابتدا بیت start داده میشود که دریافت کننده متوجه شود از اینجا کار را آغاز کند.

سپس یک بیت به نام Parity داده میشود که برای بررسی این است که آیا بیتی در این انتقال تغییر یافته است یا نه. مثلا اگر noise داشته باشیم این اتفاق می افتد و یک بیت مقدارش عوض می شود.

پس از آن، بیت های داده ورود را به صورت سریال به خروجی می دهد. (در اینجا 7 بیت) و در پایان بیت end داده می شود و کار به پایان می رسد.

بخش گیرنده:

ورودی این بخش یک سیم است که همان بیت ها (و سه سیگنال start, parity, end) به صورت سریال، ورودی داده میشود.

کار این بخش این است که آن 7 بیت داده را جدا کند و به صورت موازی خروجی بدهد. و البته اگر parity داده شده با parity محاسبه شده تفاوت داشت، یعنی داده غلط به مقصد رسیده است و میگوید که خطا داریم.

این دو ماژول در هر UART مستقل از یکدیگر هستند.

برای ایجاد یک کانال یک طرفه، دو تا UART را باید به یکدیگر متصل کنیم. نحوه اتصال هم بدین صورت است که فرستنده اولی را به گیرنده دومی وصل می کنیم.

برای ایجاد کانال دو طرفه، علاوه بر ارتباط بالا، باید فرستنده دومی را نیز به گیرنده اولی وصل کنیم.

کد:

UART.v

|  |
| --- |
| module UART(clk,  rstn,  tx\_start,  tx\_data\_in,  tx\_channel\_out,  rx\_channel\_in,  rx\_data\_out,  rx\_out\_vaild);    parameter BIT\_LEN = 7;    input clk, rstn, tx\_start, rx\_channel\_in;  input [BIT\_LEN-1:0] tx\_data\_in;  output tx\_channel\_out, rx\_out\_vaild;  output [BIT\_LEN-1:0] rx\_data\_out;    TX #(.BIT\_LEN(BIT\_LEN)) tx (  .clk(clk),  .rstn(rstn),  .start(tx\_start),  .data\_in(tx\_data\_in),  .channel\_out(tx\_channel\_out)  );    RX #(.BIT\_LEN(BIT\_LEN)) rx (  .clk(clk),  .rstn(rstn),  .channel\_in(rx\_channel\_in),  .data\_out(rx\_data\_out),  .is\_valid(rx\_out\_vaild)  );      endmodule |

این ماژول اصلی است که در آن از دو ماژول RX و TX دو تا instance ساخته ایم.

و پارامتر bit\_len را برابر با 7 قرار داده ایم.

TX.v

|  |
| --- |
| `define RST 1'b0  `define SEND 1'b1  module TX (clk,  rstn,  start,  data\_in,  channel\_out);    parameter BIT\_LEN = 7;    input clk, rstn, start;  input [BIT\_LEN - 1:0] data\_in;  output reg channel\_out;    reg state;  reg [BIT\_LEN - 1:0] buffer;  reg [$clog2(BIT\_LEN + 1 + 1 + 1):0] send\_idx;    always @(posedge clk or negedge rstn) begin  if (!rstn) begin  state <= `RST;  end  else begin  case (state)  `SEND:  begin  if (send\_idx > BIT\_LEN + 2) begin  state <= `RST;  end  else begin  send\_idx <= send\_idx + 1;  end  end  `RST:  begin  if (start) begin  state <= `SEND;  end  end  endcase  end  end    always @(state or send\_idx) begin  case (state)  `SEND:  begin  if (send\_idx == 0) begin  buffer <= data\_in;  channel\_out <= 1;  end  else if (send\_idx == 1) begin  channel\_out <= ^buffer;  end  else if (1 < send\_idx && send\_idx <= BIT\_LEN + 1) begin  channel\_out <= buffer[send\_idx - 2];  end  else if (send\_idx > BIT\_LEN + 1) begin  channel\_out <= 1;  end  end  `RST:  begin  buffer <= 0;  send\_idx <= 0;  channel\_out <= 0;  end  endcase  end    endmodule |

در ماژول TX که همان Transfer مان است، دو تا state داریم. یکی state = reset که هیچ کاری نمی کند و یک استیت state = send که داده را منتقل می کند.

این ماژول با بیت start شروع به کار میکند. و بعدی ها رو transfer میکند. و 7 بیت ورودی موازی را سریال میکنیم. (data\_in) و channel\_out همان یک سیگنال خروجی سریال است.

در always block اول (خطوط 20 تا 43) ابتدا مشخص میکنیم که در هر کلاک، متناسب با state فعلی، state بعدی را مشخص میکنیم. (اگر در حال انتقال بودیم و send\_idx به بیشتر از تعداد بیتهای مورد انتقال شد، به استیت reset برمیگردیم.)

در always block دوم (خطوط 45 تا 70) باید کاری را که در هر state انجام دهیم، مشخص کنیم.

اگر send\_idx = 0 باشد بیت start فرستاده شود.

اگر send\_idx = 1 باشد بیت parity فرستاده شود.

اگر بین 2 تا 9 باشد باید آن بیت را بفرستد و اگر 10 شد بیت end را میفرستد.

RX.v

|  |
| --- |
| `define RST 1'b0  `define RECV 1'b1  module RX (clk,  rstn,  channel\_in,  data\_out,  is\_valid);    parameter BIT\_LEN = 7;    input clk, rstn, channel\_in;  output reg is\_valid;  output reg [BIT\_LEN - 1:0] data\_out;    reg state, parity;  reg [BIT\_LEN - 1:0] buffer;  reg [$clog2(BIT\_LEN + 1 + 1 + 1):0] fetch\_idx;    always @(posedge clk or negedge rstn) begin  if (!rstn) begin  state <= `RST;  end  else begin  case (state)  `RECV:  begin  if (channel\_in && fetch\_idx > BIT\_LEN + 1) begin  state <= `RST;  end  else if (fetch\_idx < BIT\_LEN + 2) begin  fetch\_idx <= fetch\_idx + 1;  end  end  `RST:  begin  if (channel\_in) begin  state <= `RECV;  end  end  endcase  end  end    always @(state or fetch\_idx) begin  case (state)  `RECV:  begin  if (fetch\_idx == 1) begin  parity <= channel\_in;  end  else if (1 < fetch\_idx && fetch\_idx <= BIT\_LEN + 1) begin  buffer[fetch\_idx - 2] <= channel\_in;  end  else if (channel\_in && fetch\_idx == BIT\_LEN + 2) begin  data\_out <= buffer;  is\_valid <= (^buffer == parity);  end  end  `RST:  begin  buffer <= 0;  fetch\_idx <= 0;  end  endcase  end    endmodule |

در ماژول Receiver هم مشابه ماژول Transfer، دو تا always block داریم که اولی state بعدی را مشخص میکند و دومی هر کاری که در استیت فعلی باید انجام دهیم را انجام میدهد.

در بلاک اول (خط 20 تا 43) که state بعدی را ست میکنیم. اما یک نکته وجود دارد:

چون همیشه منتظر اولین سیگنال (start) است، همیشه Receiver یک کلاک از Transfer عقب تر است.

پس زمانی fetch\_idx تغییر می کند که آن طرف send\_idx تغییر کند و یک کلاک نیز رد شده باشد.

در بلاک دوم:

اگر در state = RECV باشیم، متناسب با fetch idx کارهای مختلفی میکند:

اگر fetch idx = 1 بود، parity را دریافت میکند.

اگر بین 1 و 8 بود، داده های سریال را در یک بافر نگه می دارد.

و در نهایت هم اگر fetch idx > 8 بود، سیگنال is valid را باید ست کند.

توضیح تکمیلی درباره parity:

سیگنال parity در واقع xor تمام بیت های بافر است. سیگنال is valid بررسی میکند که آیا سیگنال parity با xor داده های دریافت شده یکسان است یا خیر.

در صورتی که یکی بود، سیگنال is valid = 1 می شود.

توضیحات بافر:

بافر حالت backup دارد. که اگر احیانا ورودی تغییر کرد ما ورودی قبلی را داشته باشیم و آن را از دست ندهیم.

به محض اینکه بیت start آمد، ما ورودی را در بافر مینویسیم و شروع میکنیم.

Noise\_gen.v

|  |
| --- |
| module NOISE\_GENERATOR(clk,  sig\_in,  en,  sig\_out);    input clk, sig\_in, en;  output sig\_out;    integer counter;    assign sig\_out = (en) ? ((counter == 5) ? !sig\_in : sig\_in) : sig\_in;    always @(posedge clk) begin  if (counter < 15) begin  counter = counter + 1;  end  else begin  counter = 0;  end  end  endmodule |

testbench.v

|  |
| --- |
| `define NULL 0  module testbench();    parameter BIT\_LEN = 7;  parameter clk\_c = 10;    reg clk, u0\_rstn, u1\_rstn, u0\_start, u1\_start, ng0\_en, ng1\_en;  reg [BIT\_LEN - 1:0] u0\_in, u1\_in;  wire [BIT\_LEN - 1:0] u0\_out, u1\_out;  wire u0\_rx\_channel, u0\_tx\_channel, u1\_rx\_channel, u1\_tx\_channel, u0\_valid, u1\_vaild;    UART #(.BIT\_LEN(BIT\_LEN)) uart0(  .clk(clk),  .rstn(u0\_rstn),  .tx\_start(u0\_start),  .tx\_data\_in(u0\_in),  .tx\_channel\_out(u0\_tx\_channel),  .rx\_channel\_in(u0\_rx\_channel),  .rx\_data\_out(u0\_out),  .rx\_out\_vaild(u0\_valid)  );    UART #(.BIT\_LEN(BIT\_LEN)) uart1(  .clk(clk),  .rstn(u1\_rstn),  .tx\_start(u1\_start),  .tx\_data\_in(u1\_in),  .tx\_channel\_out(u1\_tx\_channel),  .rx\_channel\_in(u1\_rx\_channel),  .rx\_data\_out(u1\_out),  .rx\_out\_vaild(u1\_valid)  );    NOISE\_GENERATOR ng0 (  .clk(clk),  .sig\_in(u0\_tx\_channel),  .en(ng0\_en),  .sig\_out(u1\_rx\_channel)  );    NOISE\_GENERATOR ng1 (  .clk(clk),  .sig\_in(u1\_tx\_channel),  .en(ng1\_en),  .sig\_out(u0\_rx\_channel)  );    initial begin  $dumpfile("report/waveform.vcd");  $dumpvars(0,uart0, uart1);  end    initial begin  clk = 0;  forever clk = #(clk\_c/2) ~clk;  end    integer data\_file;  integer scan\_file;  integer seed;  initial begin  data\_file = $fopen("seed.dat", "r");  if (data\_file == `NULL) begin  $display("data\_file handle was NULL");  $finish;  end  scan\_file = $fscanf(data\_file, "%d", seed);  if (scan\_file == `NULL) begin  $display("integer read error");  $finish;  end  end    integer i;  integer n;  initial begin  u0\_rstn = 0;  u1\_rstn = 0;  ng1\_en = 0;  ng0\_en = 0;  #clk\_c;  u0\_rstn = 1;  u1\_rstn = 1;  n = 50;    $display("test connection uart0 -> uart1");  for (i = 0; i < n; i++) begin  u0\_start = 1;  u0\_in = {BIT\_LEN{$random(seed)}};  #clk\_c  u0\_start = 0;  while(uart1.rx.state || uart0.tx.state) begin  #clk\_c;  end  if (u0\_in == u1\_out && u1\_valid) begin  $display("#%d (u0:%b -> u1-v:%b(%b)) test passed", i, u0\_in, u1\_out, u1\_valid);  end  else begin  $display("#%d (u0:%b -> u1-v:%b(%b)) test failed", i, u0\_in, u1\_out, u1\_valid);  end  end    $display("test connection uart1 -> uart0");  for (i = 0; i < n; i++) begin  u1\_start = 1;  u1\_in = {BIT\_LEN{$random(seed)}};  #clk\_c  u1\_start = 0;  while(uart0.rx.state || uart1.tx.state) begin  #clk\_c;  end  if (u1\_in == u0\_out && u0\_valid) begin  $display("#%d (u1:%b -> u0-v:%b(%b)) test passed", i, u1\_in, u0\_out, u0\_valid);  end  else begin  $display("#%d (u1:%b -> u0-v:%b(%b)) test failed", i, u1\_in, u0\_out, u0\_valid);  end  end    n = 10;    ng0\_en = 1;  #clk\_c  $display("test connection with noise uart0 -> uart1");  for (i = 0; i < n; i++) begin  u0\_start = 1;  u0\_in = {BIT\_LEN{$random(seed)}};  #clk\_c  u0\_start = 0;  while(uart1.rx.state || uart0.tx.state) begin  #clk\_c;  end  if (u0\_in == u1\_out && u1\_valid) begin  $display("#%d (u0:%b -> u1-v:%b(%b)) test passed", i, u0\_in, u1\_out, u1\_valid);  end  else begin  $display("#%d (u0:%b -> u1-v:%b(%b)) test failed (noise detected)", i, u0\_in, u1\_out, u1\_valid);  end  end    ng1\_en = 1;  #clk\_c;  $display("test connection with noise uart0 -> uart1");  for (i = 0; i < n; i++) begin  u1\_start = 1;  u1\_in = {BIT\_LEN{$random(seed)}};  #clk\_c  u1\_start = 0;  while(uart0.rx.state || uart1.tx.state) begin  #clk\_c;  end  if (u1\_in == u0\_out && u0\_valid) begin  $display("#%d (u1:%b -> u0-v:%b(%b)) test passed", i, u1\_in, u0\_out, u0\_valid);  end  else begin  $display("#%d (u1:%b -> u0-v:%b(%b)) test failed (noise detected)", i, u1\_in, u0\_out, u0\_valid);  end  end  $finish;  end    endmodule |

توضیحات تست بنچ:

یک ماژول noise generator قرار داده ایم که در یک سری کلاک خاص می آید یک بیت را not میکند تا تغییر کند.

در تست بنچ ابتدا دو ماژول UART و دو ماژول ng0, ng1 را instantiate میکنیم.

ابتدا انتقال از uart0 به uart1 را چک میکنیم.

سپس انتقال از uart1 به uart0 را چک میکنیم.

در ادامه ورودی را از noise generator رد میکنیم که در آن noise بیاندازد و درست بودن سیگنال parity را نیز تست کنیم.

یک بار از uart0 به uart1

و بار دیگر برعکس.

برای بعضی از ورودی ها نویز ایجاد می شود که سیگنال parity این را نشان میدهد.

هرکدام از این انتقال ها را برای 50 تا ورودی بررسی میکنیم.

برای ساختن عدد رندوم هم مشابه آزمایش های قبل یک فایل .sh داریم که عدد رندوم را برایمان به کمک seed.dat تولید کند و موقع اجرا به تست بنچ داده می شود.

نتایج تست بنچ در پوشه report و فایل results.txt موجود است

Results.txt

|  |
| --- |
| VCD info: dumpfile report/waveform.vcd opened for output.  test connection uart0 -> uart1  # 0 (u0:1010001 -> u1-v:1010001(1)) test passed  # 1 (u0:1000110 -> u1-v:1000110(1)) test passed  # 2 (u0:0100001 -> u1-v:0100001(1)) test passed  # 3 (u0:1011010 -> u1-v:1011010(1)) test passed  # 4 (u0:0001001 -> u1-v:0001001(1)) test passed  # 5 (u0:1110111 -> u1-v:1110111(1)) test passed  # 6 (u0:0111111 -> u1-v:0111111(1)) test passed  # 7 (u0:1000001 -> u1-v:1000001(1)) test passed  # 8 (u0:1001010 -> u1-v:1001010(1)) test passed  # 9 (u0:1001101 -> u1-v:1001101(1)) test passed  # 10 (u0:0101100 -> u1-v:0101100(1)) test passed  # 11 (u0:0011110 -> u1-v:0011110(1)) test passed  # 12 (u0:1011110 -> u1-v:1011110(1)) test passed  # 13 (u0:0110110 -> u1-v:0110110(1)) test passed  # 14 (u0:1100000 -> u1-v:1100000(1)) test passed  # 15 (u0:0000100 -> u1-v:0000100(1)) test passed  # 16 (u0:1110100 -> u1-v:1110100(1)) test passed  # 17 (u0:0100011 -> u1-v:0100011(1)) test passed  # 18 (u0:1100101 -> u1-v:1100101(1)) test passed  # 19 (u0:0010110 -> u1-v:0010110(1)) test passed  # 20 (u0:0111001 -> u1-v:0111001(1)) test passed  # 21 (u0:1001101 -> u1-v:1001101(1)) test passed  # 22 (u0:1111011 -> u1-v:1111011(1)) test passed  # 23 (u0:1101101 -> u1-v:1101101(1)) test passed  # 24 (u0:1011101 -> u1-v:1011101(1)) test passed  # 25 (u0:0011010 -> u1-v:0011010(1)) test passed  # 26 (u0:1111001 -> u1-v:1111001(1)) test passed  # 27 (u0:1011000 -> u1-v:1011000(1)) test passed  # 28 (u0:1110111 -> u1-v:1110111(1)) test passed  # 29 (u0:0100100 -> u1-v:0100100(1)) test passed  # 30 (u0:0011010 -> u1-v:0011010(1)) test passed  # 31 (u0:1110110 -> u1-v:1110110(1)) test passed  # 32 (u0:1001000 -> u1-v:1001000(1)) test passed  # 33 (u0:0010001 -> u1-v:0010001(1)) test passed  # 34 (u0:1110000 -> u1-v:1110000(1)) test passed  # 35 (u0:1101011 -> u1-v:1101011(1)) test passed  # 36 (u0:1010011 -> u1-v:1010011(1)) test passed  # 37 (u0:0111100 -> u1-v:0111100(1)) test passed  # 38 (u0:0111101 -> u1-v:0111101(1)) test passed  # 39 (u0:0001101 -> u1-v:0001101(1)) test passed  # 40 (u0:0111111 -> u1-v:0111111(1)) test passed  # 41 (u0:1100001 -> u1-v:1100001(1)) test passed  # 42 (u0:1011011 -> u1-v:1011011(1)) test passed  # 43 (u0:1101000 -> u1-v:1101000(1)) test passed  # 44 (u0:1110110 -> u1-v:1110110(1)) test passed  # 45 (u0:1001010 -> u1-v:1001010(1)) test passed  # 46 (u0:1111010 -> u1-v:1111010(1)) test passed  # 47 (u0:1101000 -> u1-v:1101000(1)) test passed  # 48 (u0:0001101 -> u1-v:0001101(1)) test passed  # 49 (u0:0101010 -> u1-v:0101010(1)) test passed  test connection uart1 -> uart0  # 0 (u1:1100100 -> u0-v:1100100(1)) test passed  # 1 (u1:0000001 -> u0-v:0000001(1)) test passed  # 2 (u1:1010011 -> u0-v:1010011(1)) test passed  # 3 (u1:0110001 -> u0-v:0110001(1)) test passed  # 4 (u1:1111101 -> u0-v:1111101(1)) test passed  # 5 (u1:0111011 -> u0-v:0111011(1)) test passed  # 6 (u1:0010100 -> u0-v:0010100(1)) test passed  # 7 (u1:1111000 -> u0-v:1111000(1)) test passed  # 8 (u1:0111110 -> u0-v:0111110(1)) test passed  # 9 (u1:1100010 -> u0-v:1100010(1)) test passed  # 10 (u1:1111000 -> u0-v:1111000(1)) test passed  # 11 (u1:1011100 -> u0-v:1011100(1)) test passed  # 12 (u1:1100010 -> u0-v:1100010(1)) test passed  # 13 (u1:1001001 -> u0-v:1001001(1)) test passed  # 14 (u1:0000101 -> u0-v:0000101(1)) test passed  # 15 (u1:1010111 -> u0-v:1010111(1)) test passed  # 16 (u1:1110010 -> u0-v:1110010(1)) test passed  # 17 (u1:1110000 -> u0-v:1110000(1)) test passed  # 18 (u1:0101111 -> u0-v:0101111(1)) test passed  # 19 (u1:1111101 -> u0-v:1111101(1)) test passed  # 20 (u1:1111011 -> u0-v:1111011(1)) test passed  # 21 (u1:1010111 -> u0-v:1010111(1)) test passed  # 22 (u1:1110110 -> u0-v:1110110(1)) test passed  # 23 (u1:0110011 -> u0-v:0110011(1)) test passed  # 24 (u1:0010000 -> u0-v:0010000(1)) test passed  # 25 (u1:1110010 -> u0-v:1110010(1)) test passed  # 26 (u1:0000111 -> u0-v:0000111(1)) test passed  # 27 (u1:0100100 -> u0-v:0100100(1)) test passed  # 28 (u1:1101000 -> u0-v:1101000(1)) test passed  # 29 (u1:1111110 -> u0-v:1111110(1)) test passed  # 30 (u1:1011001 -> u0-v:1011001(1)) test passed  # 31 (u1:0101101 -> u0-v:0101101(1)) test passed  # 32 (u1:0111011 -> u0-v:0111011(1)) test passed  # 33 (u1:0010011 -> u0-v:0010011(1)) test passed  # 34 (u1:0111010 -> u0-v:0111010(1)) test passed  # 35 (u1:0111010 -> u0-v:0111010(1)) test passed  # 36 (u1:0011110 -> u0-v:0011110(1)) test passed  # 37 (u1:1011111 -> u0-v:1011111(1)) test passed  # 38 (u1:1010010 -> u0-v:1010010(1)) test passed  # 39 (u1:1000001 -> u0-v:1000001(1)) test passed  # 40 (u1:0011100 -> u0-v:0011100(1)) test passed  # 41 (u1:1000010 -> u0-v:1000010(1)) test passed  # 42 (u1:1110110 -> u0-v:1110110(1)) test passed  # 43 (u1:1100000 -> u0-v:1100000(1)) test passed  # 44 (u1:0111101 -> u0-v:0111101(1)) test passed  # 45 (u1:0111001 -> u0-v:0111001(1)) test passed  # 46 (u1:1110000 -> u0-v:1110000(1)) test passed  # 47 (u1:1010000 -> u0-v:1010000(1)) test passed  # 48 (u1:0101111 -> u0-v:0101111(1)) test passed  # 49 (u1:1110100 -> u0-v:1110100(1)) test passed  test connection with noise uart0 -> uart1  # 0 (u0:0000010 -> u1-v:0000011(0)) test failed (noise detected)  # 1 (u0:0100101 -> u1-v:0110101(0)) test failed (noise detected)  # 2 (u0:1010110 -> u1-v:1010110(1)) test passed  # 3 (u0:1010110 -> u1-v:0101100(1)) test failed (noise detected)  # 4 (u0:1000110 -> u1-v:1000110(1)) test passed  # 5 (u0:0010000 -> u1-v:0010000(0)) test failed (noise detected)  # 6 (u0:1110000 -> u1-v:1111000(0)) test failed (noise detected)  # 7 (u0:1001111 -> u1-v:1111000(0)) test failed (noise detected)  # 8 (u0:1000110 -> u1-v:1000110(1)) test passed  # 9 (u0:0100101 -> u1-v:0100101(0)) test failed (noise detected)  test connection with noise uart1 -> uart0  # 0 (u1:0000111 -> u0-v:0000011(0)) test failed (noise detected)  # 1 (u1:1100010 -> u0-v:0100010(0)) test failed (noise detected)  # 2 (u1:1011010 -> u0-v:1011010(1)) test passed  # 3 (u1:1010001 -> u0-v:1010001(1)) test passed  # 4 (u1:1101101 -> u0-v:1101001(0)) test failed (noise detected)  # 5 (u1:0100100 -> u0-v:1100100(0)) test failed (noise detected)  # 6 (u1:0110001 -> u0-v:0110001(1)) test passed  # 7 (u1:0000001 -> u0-v:0000001(1)) test passed  # 8 (u1:1111001 -> u0-v:1111101(0)) test failed (noise detected)  # 9 (u1:0011101 -> u0-v:1011101(0)) test failed (noise detected) |

***پایان***