Разработка контроллера протокола MIL-STD-1553B на ПЛИС. Часть 2

Дмитрий ДАЙНЕКО
dyne@micran.ru

предыдущей, первой части этой статьи было приведено подробное описание авиационного протокола MIL-STD-1553B. Рассмотрена различная элементная база, необходимая для реализации контроллера этого протокола, выбраны и обоснованы конкретные компоненты. Также проанализирована структурная схема системы управления на базе ПЛИС, о HDL-коде которой мы и расскажем во второй части статьи. Весь материал представлен максимально подробно, а значит, будет полезен начинающему разработчику. Итак, наш HDL-проект состоит из головного модуля (top-level) Top_MIL_1553B, который включает в себя три модуля следующего уровня иерархии — Transmitter, Receiever и RT_control. В этой части статьи представлен HDL-код модулей передатчика и приемника.

ВНаписание HDL-кода проекта

Мы попытаемся максимально подробно объяснить работу каждого модуля проекта. Сначала расскажем о тех модулях, которые непосредственно занимаются обработкой протокола MIL-STD-1553B, а потом уже о модулях, реализующих логику управления.

Модуль Top_MIL_1553B

Естественно, описание HDL-кода проекта целесообразнее всего начинать с модуля самого верхнего уровня (top-level). Наш модуль верхнего уровня подключает модули передатчика и приемника (Transmitter.v и Receiver.v) к модулю отработки алгоритма оконечного устройства (RT_control.v). Помимо этого, он обеспечивает функционирование обоих каналов протокола — основного и резервного.

Рассмотрим интерфейс ввода/вывода этого модуля:

```
module Top_MIL_1553B (

   input clk,
   input reset,

   //MKIO interface - channel A
   input DI1A, DI0A,
   output DO1A, DO0A,
   output RX_STROB_A,
   output TX_INHIBIT_A,

   //MKIO interface - channel B
   input DI1B, DI0B,
   output DO1B, DO0B,
   output RX_STROB_B,
   output TX_INHIBIT_B,

   //device3 RAM interface
   input [4:0] addr_rd_dev3,
   input clk_rd_dev3,
   output [15:0] out_data_dev3,
   output busy_dev3,

   //device5 RAM interface
   input [15:0] in_data_dev5,
   input [4:0] addr_wr_dev5,
   input clk_wr_dev5,
   input we_dev5,
   output busy_dev5
   );
```

Чтобы было легче ориентироваться в HDL-коде проекта, разработчик использует комментарии. Под текстом “//MKIO interface — channel A” представлены сигналы для драйвера протокола MIL-STD-1553B, соответствующие основному каналу. Сигналы DI1A, DI0A — это прямой и инверсный сигналы от приемника драйвера (рис. 8 [5]), а сигналы DO1A, DO0A — это аналогичные сигналы на передатчик драйвера. Сигналы RX_STROB_A, TX_INHIBIT_A — это сигналы управления каналом, а именно разрешение работы приемника и передатчика драйвера соответственно. Внимательно изучив официальную документацию на микросхему BU-67401L, можно выяснить, что приемник драйвера управляется прямой логикой, а значит, приемник будет активен при наличии «лог. 1» на сигнале RX_STROB_A. В то же время передатчик драйвера управляется инверсной логикой, то есть передатчик будет активен, когда на сигнале TX_INHIBIT_A будет установлен «лог. 0».

Под текстом “//MKIO interface — channel B” представлены сигналы для драйвера протокола MIL-STD-1553B, соответствующие резервному каналу. Назначение этих сигналов полностью идентично сигналам основного канала.

Под текстом комментариев “//device3 RAM interface” и “//device5 RAM interface” представлены сигналы обращения к памяти хранения данных, соответствующие субадресам (sub-address) 3 и 5. Ранее было решено, что подмодуль device3.v будет обеспечивать прием данных с магистрали протокола MIL-STD-1553B и хранить принятые данные во внутренней ОЗУ, а значит, что для некой сторонней периферии необходим доступ к этой ОЗУ для чтения принятых данных. Подмодуль device5.v должен обеспечивать передачу информации из внутренней ОЗУ в магистраль протокола, поэтому сторонняя периферия должна иметь интерфейс заполнения этой внутренней ОЗУ. Описанием всех этих сигналов целесообразнее заняться именно при рассмотрении подмодулей device3.v и device5.v.

Сначала проанализируем «тело» модуля Top_MIL_1553B. v.

В следующем фрагменте кода мы получаем тактовый сигнал частотой 16 МГц, необходимый для работы с протоколом:

```
//clocks
wire clk32 = clk;
reg clk16 = 1'b0;
always @ (posedge clk32) clk16 <= !clk16;
```

Входную частоту нашего устройства — 32 МГц — мы выбрали исходя из следующих соображений: для стабильной работы проекта системный тактовый сигнал должен превышать частоту работы Transmitter.v и Receiver.v, а также быть кратным ей.

Далее с помощью элемента «логическое ИЛИ» мы объединяем выходные линии обоих каналов — основного и резервного. Таким образом, в любой момент наше устройство

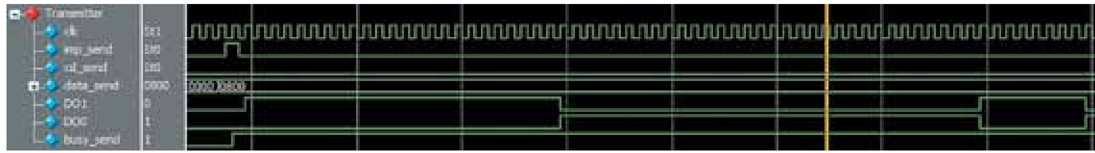

Рис. 11. Сигналы ввода/вывода модуля Transmitter.v

сможет адекватно принять входной поток информации от магистрали. Выходные же линии связи просто включены параллельно, поэтому отправляемая информация будет присутствовать на обоих каналах:

```
wire DI1, DI0, DO1, DO0;

assign DI1 = DI1A | DI1B;
assign DI0 = DI0A | DI0B;

assign DO1A = DO1;
assign DO0A = DO0;
assign DO1B = DO1;
assign DO0B = DO0;
```

Читатель может заявить, что резервирование канала организовано слишком просто. Но автор спешит напомнить, что его цель — разобраться в особенностях приема и отправки информации по данному протоколу на примере простого проекта. Как было сказано в разделе описания протокола MIL-STD-1553B, этот стандарт позволяет использовать команды управления, которые, помимо всего прочего, «умеют» блокировать или разрешать резервную линию. Поэтому логику резервирования канала можно реализовать намного сложнее, однако в данном проекте автор решил этого не делать.

Следующим текстом кода мы разрешаем/запрещаем работу приемника и передатчика в моменты передачи информации от нашего оконечного устройства на контроллер канала. Приемник драйвера должен быть разрешен всегда, кроме момента, когда нужно отправить на контроллер канала ответное или информационное слово. Передатчик драйвера, соответственно, наоборот. Для этого мы используем сигнал от модуля передатчика tx_busy (он будет рассмотрен ниже), удлиненный на несколько тактов clk16, так как строб tx_busy в модуле Transmitter.v появляется раньше, чем происходит реальная передача данных в магистраль.

```
reg [4:0] ena_reg = 5'd0;

always @ (posedge clk16 or posedge reset)
if (reset) ena_reg <= 5'd0;
else ena_reg <= {ena_reg[3:0],tx_busy};

assign RX_STROB_A = ~(|ena_reg);
assign TX_INHIBIT_A = ~(|ena_reg);
assign RX_STROB_B = ~(|ena_reg);
assign TX_INHIBIT_B = ~(|ena_reg);
```

Подключаем в проект модули передатчика и приемника:

```
Transmitter Transmitter (
  .clk (clk16),
  .reset (reset),

  .DO1 (DO1), .DO0 (DO0),

  .imp_send (tx_ready),
  .cd_send (tx_cd),
  .data_send (tx_data),
  .busy_send (tx_busy)
       );

Receiver Receiver (
  .clk (clk16),
  .reset (reset),

  .DI1 (DI1), .DI0 (DI0),

  .data_get (rx_data),
  .cd_get (rx_cd),
  .done (rx_done),
  .parity_error (parity_error)
       );
```

Назначение всех сигналов ввода/вывода модулей подробно будет описано далее, при рассмотрении самих модулей. Следует только уточнить, что здесь к модулям подключаются линии данных магистрали протокола. Также тут указаны параллельные шины данных и сигналы сервиса, которые связывают передатчик и приемник с модулем RT_control.v, который, в свою очередь, необходимо подключить к нашему проекту:

```
RT_control RT_control (
  .clk (clk32),
  .reset (reset),

  .rx_done (rx_done),
  .rx_data (rx_data),
  .rx_cd (rx_cd),
  .p_error (parity_error),

  .tx_ready (tx_ready),
  .tx_data (tx_data),
  .tx_cd (tx_cd),
  .tx_busy (tx_busy),

  .addr_rd_dev3 (addr_rd_dev3),
  .clk_rd_dev3 (clk_rd_dev3),
  .out_data_dev3 (out_data_dev3),
  .busy_dev3 (busy_dev3),

  .in_data_dev5 (in_data_dev5),
  .addr_wr_dev5 (addr_wr_dev5),
  .clk_wr_dev5 (clk_wr_dev5),
  .we_dev5 (we_dev5),
  .busy_dev5 (busy_dev5)
       );
```

Подключение этого модуля показано на рис. 9 из [5].

Модуль Transmitter.v

Назначение этого модуля — по некоторому внешнему сигналу принять параллельные данные и последовательно передать их в виде манчестерского кода в соответствии с протоколом MIL-STD-1553B.

Рассмотрим интерфейс ввода/вывода модуля:

```
module Transmitter (
  input clk,
  input reset,

  input imp_send,
  input cd_send,
  input [15:0] data_send,
  output reg busy_send,

  output reg DO1, DO0
  );
```

Назначение тактового сигнала clk и внешнего сигнала сброса reset объяснять не требуется. Нужно только уточнить, что сигнал clk должен иметь частоту 16 МГц, чтобы обеспечить протокольную частоту обмена данными в 1 МГц. По импульсу imp_send параллельные данные будут «защелкнуты», и начнется передача манчестерского кода по дифференциальным линиям DO1 и DO0. Напомним, что существует три разных слова — командное, информационное и ответное. Если нужно передать командное или ответное слово, используется один синхросигнал, если информационное — другой. Так вот именно сигнал cd_send и сообщает модулю, какой синхросигнал использовать. Параллельная 16-разрядная шина data_send — это передаваемые данные. Если необходимо передать командное слово, то на этой шине должны присутствовать адрес, субадрес, количество слов и бит приема-передачи. Если же необходимо передать информационное слово, то, соответственно, 16 бит данных. Наличие «лог. 1» на сигнале busy_send говорит о том, что идет передача. Временные диаграммы этих сигналов приведены на рис. 11.

Перейдем к описанию сигналов, используемых в этом модуле:

```
reg [15:0] data_buf;
reg cd_buf;
reg [2:0] length_bit;
reg [5:0] count_bit;

wire [31:0] data_manchester;
wire [39:0] word_manchester;
wire parity;
```

Сигнал cd_buf и 16-разрядный data_buf — это «защелкнутый» бит нужного синхросигнала и шина данных соответственно. 32-разрядный data_manchester — это, проще говоря, удвоенный 16-разрядный сигнал data_send. Из раздела описания протокола читатель узнал, что в манчестерском коде каждый бит данных представляет со-

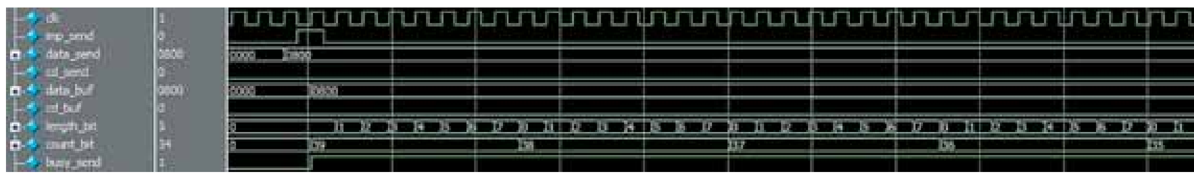

Рис. 12. Временные диаграммы модуля Transmitter.v

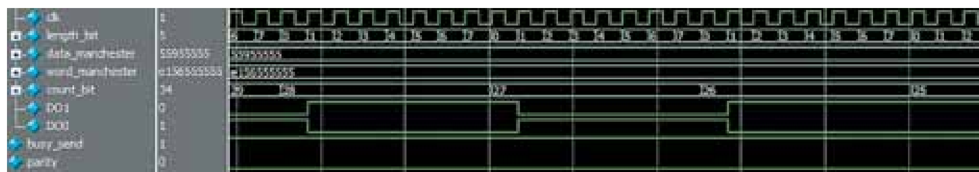

Рис. 13. Временные диаграммы модуля Transmitter.v

бой переход с «лог. 0» в «лог. 1» или наоборот — с «лог. 0» в «лог. 1». Поэтому data_manchester и имеет удвоенную разрядность. 40-разрядный word_manchester — это полностью манчестерское слово, содержащее помимо data_manchester еще синхросигнал и паритет (удвоенные 20 бит передаваемого слова). Что касается 3-разрядного length_bit, то в момент переполнения этого счетчика на выходе появляется следующий разряд сигнала word_manchester [39:0]. Другими словами, регистр length_bit — счетчик длины (в тактах clk) элемента манчестерского кода. А вот счетчик count_bit — это уже номер элемента манчестерского кода. Когда начнем разбирать код модуля, понять все будет намного проще.

Рассмотрим следующую часть модуля:

```
1.always @ (posedge clk or posedge reset)
2.begin
3.    if (reset) begin
4.        busy_send <= 1'b0;
5.        data_buf <= 16'd0;
6.        cd_buf <= 1'b0;
7.        length_bit <= 3'd0;
8.        count_bit <= 6'd0;
9.    end
10.   else begin
11.
12.       if (imp_send) begin
13.           data_buf <= data_send;
14.           cd_buf <= cd_send; end
15.
16.       if (imp_send) length_bit <= 3'd0;
17.       else if (busy_send) length_bit <= length_bit + 1'b1;
18.
19.   if (imp_send) count_bit <= 6'd39;
20.       else if ((count_bit != 6'd0) & (length_bit == 3'd7))
21.           count_bit <= count_bit - 1'b1;
22.
23.       if (imp_send) busy_send <= 1'b1;
24.       else if ((count_bit == 6'd0) & (length_bit == 3'd7))
25.           busy_send <= 1'b0;
26.
27.   end
28.end
```

В строках 3–10 по сигналу сброса reset происходит начальная установка всех сигналов этого процесса в начальное состояние. В дальнейшем подобную часть кода (начальный сброс) автор упоминать не будет, чтобы не повторяться.

В строках 12–14 описывается «защелкивание» входных данных для передачи при появлении импульса imp_send.

В строках 16–17 происходит подсчет длительности элемента манчестерского кода. Инкремент length_bit разрешается только во время передачи и сбрасывается в ноль во время «защелкивания» входных данных для передачи.

В строках 19–21 описан декремент номера элемента последовательности манчестерского кода count_bit [5:0]. Декремент происходит до нуля и только во время переполнения счетчика длины элемента length_bit [3:0].

Процесс передачи сопровождается сигналом busy_send (строки 23–25): во время «защелкивания» входных данных этот сигнал поднимается в «лог. 1», а сбрасывает в «лог. 0» при окончании передачи последнего элемента манчестерского кода. Временные диаграммы этих сигналов представлены на рис. 12.

Теперь нужно преобразовать «защелкнутые» входные данные в выходную посылку. Как мы помним, посылка состоит из синхросигнала, поля данных и бита четности. Сначала преобразуем поле данных в манчестерский вид:

```
genvar i;
generate for (i = 0; i < 16; i = i + 1)
   begin : gen_manchester
      assign data_manchester[2*i] = ~data_buf[i];
      assign data_manchester[2*i + 1] = data_buf[i];
   end
endgenerate
```

Здесь использована конструкция generate for, которая размножает нужное количество раз то, что описано в теле конструкции. Напомним, что шина data_manchester [31:0] имеет удвоенную разрядность по отношению к «защелкнутым» входным данным data_buf [15:0], потому что нам нужно иметь переходы либо с «лог. 0» в «лог. 1», либо с «лог. 1» в «лог. 0».

Далее сформируем полностью манчестерскую посылку:

```
assign word_manchester[39:34] = (cd_buf) ? 6'b000111 : 6'b111000;
assign word_manchester[33:2] = data_manchester;
assign word_manchester[1:0] = (parity) ? 2'b10 : 2'b01;
```

В первой строке кода идет выбор синхросигнала. Во второй строке присваивается только что преобразованное поле данных. В третьей же строке присваивается пара элементов манчестерского кода (или, как было упомянуто, — переход, который состоит из пары элементов), в зависимости от бита паритета. Напомним, бит паритета выбирается таким, чтобы общее количество единиц 20-разрядного слова (рис. 5 [5]) было нечетным.

Следующее выражение вычисляет нужный нам бит паритета:

```
assign parity = ~(^data_buf);
```

В завершение приведем процесс, в котором по тактовому сигналу на дифференциальные выходы модуля выставляется тот разряд посылки, готовой на передачу, номер которой указан в регистре count_bit [5:0]:

```
always @(posedge clk)
begin
   if (busy_send) begin
      DO1 <= word_manchester[count_bit];
      DO0 <= ~word_manchester[count_bit]; end
   else begin
      DO1 <= 1'b0;
      DO0 <= 1'b0; end
end
```

Когда передача отсутствует, в посылке на дифференциальных линиях должны находиться логические нули. Это необходимо для того, чтобы не нагружать линию (рис. 13).

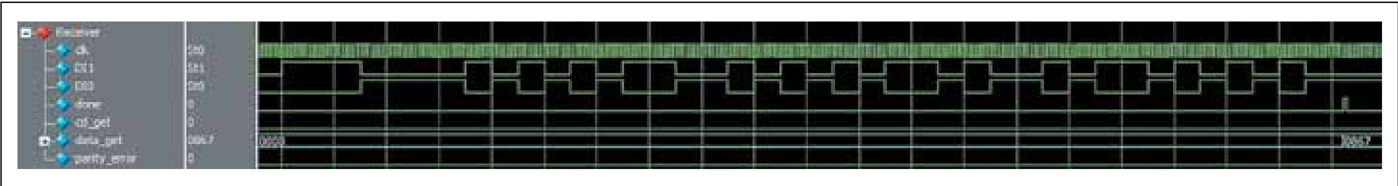

Рис. 14. Временные диаграммы модуля Receiver.v

Модуль Receiver.v

Назначение этого модуля — принять информацию по протоколу MIL-STD-1553B, проверить корректность принятой посылки, определить, какое слово принято (командное или информационное), и выставить слово в параллельном виде на выход.

Рассмотрим интерфейс ввода/вывода этого модуля:

```
module Receiver (
  input clk,
  input reset,

  input DI1, DI0,

  output reg [15:0] data_get,
  output reg cd_get,
  output reg done,
  output reg parity_error
      );
```

Здесь тактовый сигнал clk, так же как и в передатчике Transmitter.v, должен иметь частоту 16 МГц, чтобы адекватно принять входящие данные с протокольной частотой 1 МГц. Входной манчестерский код поступает на дифференциальные входы DI1, DI0. После окончания приема данных будет создан импульс done, по которому принятые данные появятся на линиях data_get [15:0]. По сигналу cd_get можно будет определить, какое слово принято — информационное или командное. Сигнал parity_error покажет, что правило паритета не было выполнено (рис. 14).

Начнем описывать тело модуля. Так как принимаемый сигнал асинхронен по отношению к нашему устройству, необходимо выполнить синхронизацию:

```
reg [2:0] pos_shift_reg;
reg [2:0] neg_shift_reg;

always @ (posedge clk or posedge reset)
begin
  if (reset) begin
    pos_shift_reg <= 3'd0;
    neg_shift_reg <= 3'd0; end
  else begin
    pos_shift_reg <= {pos_shift_reg[1:0], DI1};
    neg_shift_reg <= {neg_shift_reg[1:0], DI0}; end
end
```

Для этого используются 3-разрядные сдвиговые регистры.

Только во время обмена информацией по магистрали линии DI1 и DI0 будут находиться в противоположных друг относительно друга состояниях. Во время простоя магистрали линии будут иметь одинаковое состояние — нулевое. Следующий процесс позволяет детектировать начало приема информации:

```
reg det_sig;

always @ (posedge clk or posedge reset)
begin
  if (reset)
    det_sig <= 1'b0;
  else case ({pos_shift_reg[2], neg_shift_reg[2]})
      2'b00: det_sig <= ~det_sig;
      2'b01: det_sig <= 1'b0;
      2'b10: det_sig <= 1'b1;
      2'b11: det_sig <= ~det_sig;
  endcase
end
```

В момент простоя магистрали сигнал det_sig будет постоянно инвертироваться, что не позволит в дальнейшем начать прием слова.

Далее рассмотрим фрагмент кода, который будет фиксировать момент начала приема элемента манчестерского кода:

```
reg [2:0] sig_shift_reg;
wire in_data;
wire reset_length_bit;

always @ (posedge clk or posedge reset)
begin
  if (reset) sig_shift_reg <= 3'd0;
  else sig_shift_reg[2:0] <= {sig_shift_reg[1:0], det_sig};
end

assign in_data = sig_shift_reg[2];
assign reset_length_bit = sig_shift_reg[2] ^ sig_shift_reg[1];
```

Сначала сигнал det_sig задвигается в регистр sig_shift_reg [2:0]. «Логическое исключающее ИЛИ» из двух старших разрядов регистра позволит обнаружить момент перехода входного манчестера с «лог 0» в «лог. 1» либо наоборот. Вспомним, что код Манчестер-II, используемый в протоколе MIL-STD-1553B, — самосинхронизируется именно потому, что каждый элемент информации передается переходом с одного логического состояния в другое. На этот переход и нужно опираться. Сигнал reset_length_bit — импульс сброса счетчика длины элемента манчестерского кода, который используется в следующем процессе. Описанные процессы представлены на рис. 15.

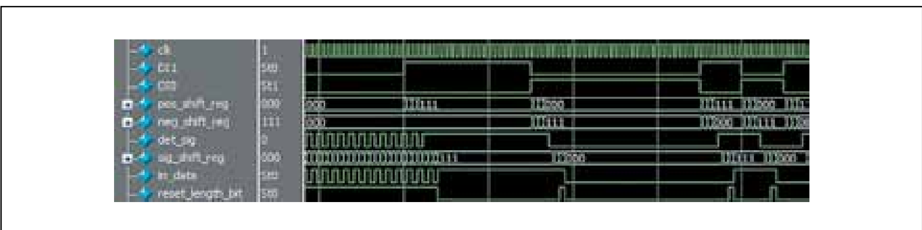

Рис. 15. Временные диаграммы модуля Receiver.v

За счет того, что наш модуль работает на тактовой частоте 16 МГц, восемь тактов сигнала синхронизации приходится на один элемент манчестерского кода. Зная момент начала приема элемента и имея счетчик длины элемента length_bit [2:0], мы можем поймать середину элемента. Следующий процесс собирает весь манчестерский код в сдвиговый регистр manchester_reg [39:0]:

```
reg [2:0] length_bit;
wire middle_bit;
reg [39:0] manchester_reg;

assign middle_bit = (length_bit == 3'd3);

always @ (posedge clk or posedge reset)
begin

  if (reset) begin
    length_bit <= 3'd0;
    manchester_reg <= 40'd0; end
  else begin
    if (reset_length_bit) length_bit <= 3'd0;
    else length_bit <= length_bit + 1'b1;

    if (middle_bit)
    manchester_reg[39:0] <= {manchester_reg[38:0], in_data};

  end
end
```

Здесь мы начинаем инкремент счетчика length_bit [2:0] сразу после обнаружения начала элемента. Когда сигнал middle_bit находится в «лог. 1» (а это произойдет в середине элемента), информация сдвигается в младший разряд регистра manchester_reg [39:0].

Следующий фрагмент кода выделяет из регистра manchester_reg сигнал паритета parity_buf, вид синхросигнала cd_buf и 16-разрядное поле данных data_buf [15:0]. Так же вырабатывается импульс правильности принятого манчестерского кода true_data_manchester:

```
wire true_data_packet;
wire [15:0] data_buf;
wire parity_buf;
wire cd_buf;

assign true_data_packet =
      ((manchester_reg[39:34] == 6'b000111) |
      (manchester_reg[39:34] == 6'b111000))
      & (manchester_reg[33] ^ manchester_reg[32])
      & (manchester_reg[31] ^ manchester_reg[30]);
```

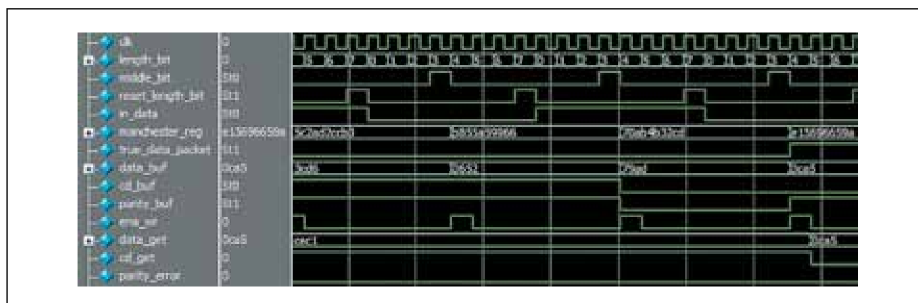

Рис. 16. Временные диаграммы модуля Receiver.v

```
genvar i;
generate for (i = 0; i <= 15; i = i + 1)
begin : bit_a
   assign data_buf[i] = manchester_reg[2*i+3];
end
         endgenerate

assign parity_buf = manchester_reg[1];
assign cd_buf = manchester_reg[35];
```

Напомним, что каждый разряд принятой информации состоит из двух элементов манчестерского кода (рис. 3 [5]). В соответствии с этим из регистра manchester_reg выделяются вид синхросигнала и бит паритета. Так же, но с использованием конструкции generate for, выделяется поле данных data_buf [15:0]. Сигнал true_data_buf устанавливается в «лог. 1» только тогда, когда в сдвиговом регистре manchester_reg появятся один из двух синхросигналов и хотя бы два верных бита информации. Можно, конечно, проверять полностью весь регистр, но, как показывает практика, двух битов достаточно.

Завершающий фрагмент кода модуля Receiver.v по окончании приема выставляет на выход принятые данные, а также вырабатывает импульс окончания приема done:

```
reg ena_wr;

always @ (posedge clk or posedge reset)
begin
   if (reset) begin
      cd_get <= 1'b0;
      parity_error <= 1'b0;
      ena_wr <= 1'b0;
      data_get <= 16'd0;
   end
   else begin

      ena_wr <= middle_bit;

      if (true_data_packet & ena_wr) begin
      cd_get <= cd_buf;
      data_get <= data_buf[15:0];
      parity_error <= ~(^({data_buf,parity_buf}));
   end

   done <= true_data_packet & ena_wr;

   end
end
```

Сигнал ena_wr — это, по сути, импульс середины элемента, задержанный на один такт clk. Он необходим, чтобы по окончании приема все данные успели задвинуться в manchester_reg и «защелкнуться» на выходных регистрах. Сигнал parity_error устанавливается в «лог. 1», если принятые данные имеют четное количество единиц, а значит, не соответствуют правилу паритета. Это показано на рис. 16.

Мы рассказали, как реализовать кодирование и декодирование последовательных слов данных в стандарте MIL-STD-1553B. Теперь осталось реализовать логику управления и формирования нужных пакетов данных.

В следующей, третьей части статьи мы рассмотрим модуль RT_control и входящие в его состав подмодули device3 и device5, которые реализуют работу устройства с субадресами 3 и 5 соответственно. ■

Литература

1. ГОСТ Р 52070-2003. «Интерфейс магистральный последовательный системы электронных модулей».
2. Дайнеко Д. Реализация CORDIC-алгоритма на ПЛИС // Компоненты и технологии. 2011. № 12.
3. IEEE Standart Verilog Hardware Description Language. 2001.
4. Mentor Graphics. ModelSim Tutorial. May, 2008.
5. Дайнеко Д. Разработка контроллера протокола MIL-STD-1553B на ПЛИС. Ч. 1 // Компоненты и технологии. 2013. № 12.