Разработка контроллера протокола MIL-STD-1553B на ПЛИС. Часть 4

Дмитрий ДАЙНЕКО
dyne@micran.ru

В предыдущей, третьей части статьи автор завершил рассмотрение HDL-кода проекта контроллера протокола MIL-STD-1553B. Был проанализирован модуль RT_control и приведены временные диаграммы. Теперь нам осталось провести моделирование HDL-проекта с использованием тестбенча, чтобы убедиться в работоспособности созданного проекта. Моделирование мы будем проводить в известной fpga-дизайнерам САПР ModelSim Altera Starter Edition.

Моделирование HDL-кода проекта в ModelSim

Используя ModelSim, мы сможем моделировать работу устройства, создав функциональный тестбенч для нашего проекта. Как должен быть реализован наш тестбенч-файл?

Первое. Тестбенч должен формировать пакеты протокола MIL-STD-1553B с адресом 1 и субадресом 3. После этого необходимо считать принятые данные из памяти ОЗУ.

Второе. Первоначально следует записать данные в ОЗУ с субадресом 5. После этого в соответствии с протоколом MIL-STD-1553B считать данные.

Главное, все действия, осуществляемые в ходе моделирования, должны выводиться на консоль, чтобы мы могли сделать выводы о работоспособности нашего HDL-проекта.

Таким образом, мы будем рассматривать HDL-проект в качестве черного ящика, устройство которого нас не интересует. Нам важен отклик системы на входные воздействия. Такой подход к созданию тестбенча оправдан в том случае, когда HDL-проект периодически модернизируется, создаются различные его версии, и малейшее изменение в коде может привести к его частичной неработоспособности. В задачу такого тестбенча входит именно автоматическая проверка откликов системы.

Модуль tb.v

Для начала подключим головной модуль HDL-проекта:

```
`timescale 1ns/1ps;
module tb ();

reg clk;
reg reset;

//1553B - channel A
reg DI1A, DI0A;
wire DO1A, DO0A;
wire RX_STROB_A;
wire TX_INHIBIT_A;

//1553B - channel B
reg DI1B, DI0B;
wire DO1B, DO0B;
wire RX_STROB_B;
wire TX_INHIBIT_B;

//MEM DEV 3 interface
reg [4:0] addr_rd_dev3;
reg clk_rd_dev3;
wire [15:0] out_data_dev3;
wire busy_dev3;

//MEM DEV 5 interface
reg [15:0] in_data_dev5;
reg [4:0] addr_wr_dev5;
reg clk_wr_dev5;
reg we_dev5;
wire busy_dev5;

Top_MIL_1553B Top_MIL_1553B(
   clk, reset,

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

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

   //Memories interface
   addr_rd_dev3, clk_rd_dev3, out_data_dev3, busy_dev3,
   in_data_dev5, addr_wr_dev5, clk_wr_dev5, we_dev5, busy_dev5
      );
```

Директивой timescale мы указали единицу временного контроля (1 нс) и разрешение по времени при симуляции (1 пс).

Далее приводится описание сигналов, которое является интерфейсом ввода/вывода top-level модуля. Сигналы, являющиеся входными для Top_MIL_1553B.v, приведены как регистры. Таким образом, мы получаем возможность присваивать им значения.

Подключение самого модуля Top_MIL_1553B.v объяснять не требуется.

В тестбенч-модуле можно использоваться несинтезируемые конструкции языка описания аппаратуры, что увеличивает возможности моделирования проектов. Использование несинтезируемых конструкций максимально приближает Verilog к обычному языку программирования.

В нашем тестбенче мы будем использовать процедуры, которые позволят объединить повторяющиеся моменты моделирования, а также более грамотно оформить модуль с точки зрения программирования.

Процедура инициализации массивов данных, используемых при моделировании:

```
reg [15:0] tb_array_dev3 [0:31];
reg [15:0] tb_array_dev5 [0:31];

task array_init;
   integer i;
   begin
      $display(" | data test dev3 | data test dev5 ");
      $display("********************|*************");
      for (i = 0; i <= 31; i = i + 1)
         begin
            tb_array_dev3[i] = {$random} % (2**16-1);
            tb_array_dev5[i] = {$random} % (2**16-1);
            $display("%0d|%h\t\t|%h\t\t",i,tb_array_dev3[i], tb_array_dev5[i]);
         end
   end
endtask
```

Как видим, процедуры на языке Verilog описываются конструкцией ***task…endtask***. Следует обратить внимание на то, что локальные переменные и сигналы объявляются до ключевого слова ***begin***. В этой процедуре есть только одна переменная i-типа — integer. С помощью системной задачи $display мы вводим в консоль шапку таблицы с 16-разрядными данными, которые и будем использовать при моделировании.

В цикле ***for***(;;) идет заполнение массивов случайными 16-разрядными данными в диапазоне от 0 до 2^{16}. Случайное число мы получаем, применяя системную функцию $random. Следует отметить, что если б мы записали функцию $random без фигурных скобок, то получили бы знаковый диапазон случайных значений — от -2^{16} до $+2^{16}$.

Последней системной функцией $display мы выводим на экран 16-разрядные значения для каждого из субадресов. Читателя могу смутить различные «иероглифы», использованные в этой функции. Но это всего

лишь атрибуты форматирования. Мы выводим на экран три аргумента. Так вот значение i будет отображаться в десятичном виде (атрибут %d), а значения обоих массивов — в шестнадцатеричном виде (атрибут %h). Также здесь присутствует атрибут табуляции — \t.

Немного опережая события, покажем, как будет выглядеть консоль после процедуры array_init (рис. 18).

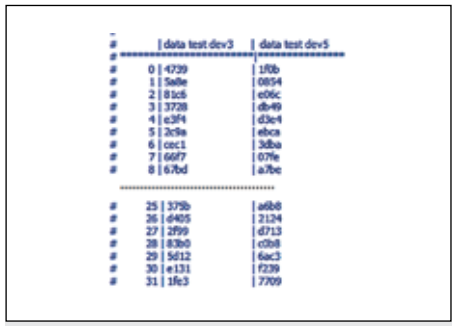

Рис. 18. Результат выполнения array_init

Процедура отправки слова в соответствии с протоколом MIL-STD-1553B:

```
task word_transmit
   (input sync, input [15:0] data);
   integer i;
   reg [39:0] shift_reg;
   reg parity;
   begin

      //set sync
      if (sync) shift_reg[39:34] = 6'b111000;
      else shift_reg[39:34] = 6'b000111;

      //set field
      for(i=0; i<=15; i=i+1)
         shift_reg[(i*2+2) +: 2] = {data[i],~data[i]};

      //set parity
      parity = 1'b1;
      for(i = 0; i<= 15; i = i + 1)
      begin
         if (data[i] == 1'b1) parity = ~parity;
         else parity = parity;
      end
      shift_reg[1:0] = {parity,~parity};

      //send
      for(i = 0; i <= 39; i = i + 1)
      begin
         DI1A = shift_reg[39-i];
      DI0A = ~shift_reg[39-i];
      #500;
      end

      DI1A = 0;
      DI0A = 0;
   end
endtask
```

Эта процедура должна выдавать на дифференциальные входы основного канала (DI1A и DI0A) последовательность манчестерского кода с правилами, рассмотренными в разделе описания протокола. Одним из отличий word_transmit от array_init является наличие двух аргументов процедуры. Аргумент sync сообщает процедуре, какой синхросигнал использовать (1 — командное слово, 0 — информационное слово). Аргумент data[15:0] сообщает процедуре, что именно нужно передать (на рис. 5 — биты 4–19).

В теле процедуры под комментарием //set sync в сдвиговый регистр загружается синхросигнал.

Под текстом комментария //set field мы преобразовываем поле данных для передачи в манчестерский код. (Вспомним, что каждый бит информации в манчестере кодируется двумя противоположными уровнями.) Здесь читатель может удивиться, увидев странную запись адресации разрядов регистра shift_reg[(i*2+2)+: 2]. Обычная запись адресации части регистра shift_reg[a:b] применяется лишь в том случае, когда a и b являются константами или не требуется работать с отрезками вектора. В нашем же случае используется выражение, и придется применить синтаксис shift_reg[a+:b], где a — младший разряд нужной нам части регистра, b — ширина нужной нам части регистра. Например, shift_reg[3+:2] <= 2'b11 эквивалентно shift_reg[4:3] <= 2'b11.

Нам осталось определить бит паритета посылки, чтобы заполнить младшие два разряда регистра shift_reg. Под текстом комментария //set parity приведен расчет бита паритета (чтобы общее количество битов в поле данных плюс сам бит паритета было нечетным).

Наконец, нам осталось передать получившийся манчестерский код на дифференциальные входы основного канала DI1A и DI0A. Это реализует цикл ***for*(;;)** под текстом комментария //send. Через каждые 500 нс цикл считывает регистр shift_reg[39:0] от старшего разряда к младшему.

Процедура приема слов:

```
task word_receiver;
   integer i;
   reg [39:0] tb_man_reg;
   reg [19:0] tb_data_reg;
   begin

      //input
      for(i = 39; i >= 0; i = i — 1)
      begin
         tb_man_reg[i] = DO1A;
         #500;
      end

      //decode
      for(i = 0; i <= 19; i = i + 1)
         tb_data_reg[i] = tb_man_reg[i*2+1];

      //indicate
      if(tb_man_reg[39:34] == 6'b111000)
         $display("Received Status Word. ADDRESS = %d, status bits = %b",
tb_data_reg[16:12], tb_data_reg[11:1]);
      else if (tb_man_reg[39:34] == 6'b000111)
         $display("Received Data Word. DATA = %h", tb_data_reg[16:1]);

   end
endtask
```

Под текстом комментария //input происходит накопление данных с частотой, соответствующей скорости обмена данных протокола. Так как выход данных нашего устройства дифференциальный, то в тестбенч-модуле целесообразнее работать с позитивной составляющей дифференциального сигнала.

Далее под текстом комментария //decode проводится декодирование манчестерского кода, накопленного в регистре tb_man_reg[39:0].

Итак (код процедуры под комментарием //indicate), принятое слово необходимо распознать либо как ответное, либо как информационное, которые отличаются (как мы знаем из раздела описания протокола) синхросигналом. Если принято ответное слово (синхросигнал SYNC C на рис. 4), то системной задачей $display выводим на консоль соответствующее сообщение со значением ADDRESS в десятичном виде (%d, tb_data_reg [16:12]) и набором битов статуса в бинарном виде (%b, tb_data_reg[11:1]). Если же принято информационное слово, то выводим соответствующее сообщение на консоль с полем данных в шестнадцатеричном виде (%h, tb_data_reg[16:1]).

На рис. 19 показано, как будет выглядеть консоль после отработки процедуры word_receive.

Рис. 19. Результат выполнения word_receive

Нам осталось создать процедуры записи и чтения внутренней памяти RAM:

```
task read_ram3
   (input [4:0] addr);
   begin
      addr_rd_dev3 = addr;
      #31.25 clk_rd_dev3 = 1'b1;
      #31.25 clk_rd_dev3 = 1'b0;
   end
endtask

task write_ram5
   (input [15:0] data, input [4:0] addr);
   begin
      in_data_dev5 <= data;
      addr_wr_dev5 <= addr;
      we_dev5 <= 1'b1;
      #31.25 clk_wr_dev5 = 1'b1;
      #31.25 clk_wr_dev5 = 1'b0;
      we_dev5 <= 1'b0;
   end
endtask
```

Закончив с объявлением процедур, мы можем перейти непосредственно к ходу моделирования.

Нашему устройству необходимо тактирование и аппаратный сброс:

```
initial
begin
   clk = 0;
   #15.625 forever #15.625 clk = !clk;
end

initial
begin
   reset = 1;
   repeat (10) @(posedge clk);
   reset = 0;
end
```

Все блоки initial выполняются параллельно. В первом блоке мы генерируем тактовый сигнал с частотой 32 МГц (половина периода равна 15,625 нс). Во втором блоке генерируется положительный импульс сброса reset. Здесь использован несинтезируемый (большинством синтезаторов) оператор repeat, который в данном примере выполняется в течение первых 10 тактов clk.

В следующем блоке initial будет показан основной ход моделирования с использованием процедур, которые мы объявили ранее.

Итак, нам нужно смоделировать работу нашего устройства, осуществив обмен данными по протоколу MIL-STD-1553B с субадресами 3 и 5. Ход моделирования можно разбить на этапы:

- инициализация входных сигналов проекта;
- инициализация массивов данных для моделирования;
- отправка командного слова с субадресом 3 и семи информационных слов;
- чтение внутренней памяти RAM субадреса 3 с выводом данных на консоль;
- запись во внутреннюю память RAM субадреса 5 с выводом данных на консоль;
- отправка командного слова с субадресом 5.

Приведем блок initial полностью:

```
integer i;

initial
begin

   //init input signals
   {DI1A, DI0A} = {2'b00};
   {DI1B, DI0B} = {2'b00};
   addr_rd_dev3 = 5'd0;
   clk_rd_dev3 = 1'b0;
   in_data_dev5 = 16'd0;
   addr_wr_dev5 = 5'd0;
   clk_wr_dev5 = 1'b0;
   we_dev5 = 1'b0;

   //test data init
   $display("\n");
   $display("*************************");
   $display("*** TEST DATA INIT ***");
   $display("*************************");
   array_init;

   #10000; //wait 10 us

   //packet for subaddr 3
   $display("\n");
   $display("*************************");
   $display("*** TESTING SUBADDR 3 ***");
   $display("*************************");
   word_transmit (1,{5'd1,1'b0,5'd3,5'd7});
   $display($time," Transmitted Command Word — ADDRESS 1,
SUBADDRESS 3, WORD 7");
   for (i=0; i<7; i=i+1)
   begin
      word_transmit(0,tb_array_dev3[i]);
      $display($time," Transmitted Data Word - DATA %h", tb_array_
dev3[i]);
   end

   #30000; //wait 30 us

   //read mem_dev3
   $display("\n");
   for (i=0; i<7; i=i+1)
   begin
      read_ram3(i);
      $display("Read MEM_DEV3, addr = %d, data = %h", i, out_
data_dev3);
   end

   #10000; //wait 10 us

   $display("\n");
   $display("*************************");
   $display("***TESTING SUBADDR 5 ***");
   $display("*************************");

   //write mem_dev5
   for (i=0; i<5; i=i+1)
   begin
      write_ram5(tb_array_dev5[i],i);
      $display("Write MEM_DEV5, addr = %d, data = %h", i, tb_
array_dev5[i]);
   end

   //packet for subaddr 5
   word_transmit (1,{5'd1,1'b1,5'd5,5'd5});
end
```

Под текстом комментария //init input signals приводится назначение начальных состояний входных портов модуля Top_MIL_1553B. Это сделано для того, чтобы в окне симуляции **Wave** мы не видели неопределенных значений.

Под комментарием //test data init на консоль выводится соответствующая надпись для наглядности моделирования. Далее идет вызов процедуры array_init, рассмотренной ранее. Кстати, в системной задаче $display у нас впервые применен атрибут форматирования \n, означающий переход на следующую строку.

Под //packet for subaddr 3 представлена последовательность вызова процедур word_transmit. Сначала выводится на консоль заголовок, говорящий о том, что мы начинаем моделировать субадрес 3. Следующий вызов процедуры word_transmit имеет аргументы (1,{5'b1, 1'b0, 5'd3, 5'd7}). Первый аргумент с «1» означает, что мы собираемся передать командное слово. Вторым аргументом мы указываем адрес ADDRESS (5'd1), признак передачи WR (1'b0), субадрес SUBADDR (5'd3) и количество передаваемых слов (5'd7). В цикле ***for*(;;)** происходит передача семи информационных слов одно за другим, которые содержат данные, созданные с помощью процедуры array_init ранее.

Далее нам нужно считать внутреннюю память и вывести данные на консоль (текст кода под комментарием //read mem dev3). Тем самым мы можем проверить правильность работы описанного в HDL-коде алгоритма приема данных от контроллера канала оконечному устройству по протоколу MIL-STD-1553B.

Теперь нам осталось промоделировать прием данных от оконечного устройства к контроллеру канала. Как мы помним, для таких целей мы «зарезервировали» субадрес 5. Для начала заполним внутреннюю память RAM (текст кода под комментарием //write mem dev5) данными в количестве пяти 16-разрядных слов, созданных ранее с помощью функции array_init.

Затем, чтобы инициировать поток принимаемой информации (ответное слово + пять информационных слов) от оконечного устройства, которым является наш HDL-проект, необходимо отправить командное слово с аргументами (1,{5'd1,1'b1,5'd5,5'd5}). Первым аргументом является признак командного слова, второй аргумент состоит из адреса оконечного устройства ADDRESS (5'd1), признака приема WR (1'b1), субадреса SUBADDR (5'd5) и количества принимаемых слов (5'd5). Отправка командного слова приводится под комментарием //packet for subaddr 5.

В блоке initial мы организовывали передачу данных от контроллера канала (как понял читатель, наш тестбенч выполняет именно функцию контроллера канала) на оконечное устройство и фиксировали происходящее на консоли. Теперь нам осталось «научиться» ловить ответные данные от оконечного устройства и выводить их на консоль, чтобы проверить работоспособность проекта.

Приведем еще один блок initial, который будет вызывать процедуру word_receiver в нужный момент времени:

```
initial
   forever
      if (DO1A ^ DO0A) begin
         word_receiver;
      end
      else begin
         @clk;
      end
```

В то время, когда магистраль не активна, драйвер протокола не должен быть нагружен, поэтому дифференциальные сигналы DO1A и DO0A имеют противоположное значение только в момент передачи данных от оконечного устройства к контроллеру канала. Именно в этот момент запускается процедура word_receiver, которая будет фиксировать информацию от оконечного устройства и выводить ее на консоль.

На рис. 20 приведена информация, выведенная на консоль в результате моделирования нашего проекта.

Рис. 20. Консоль в результате моделирования

Таким образом, просто запустив моделирование, можно проверить работоспособность проекта полностью, сравнивая ту информацию, которая отправляется в устройство, с той, которая принимается от него.

Запустить процесс моделирования можно с помощью GUI-интерфейса ModelSim. Этот способ подробно был рассмотрен в [2]. В этом же источнике показан более рациональный способ запуска моделирования и оформление временных диаграмм — с использованием do-файла. В таком файле прописываются скрипт-команды на языке tcl, которые позволя-

ют компилировать проект, запускать симулятор и оформлять окно **Wave** симулятора автоматически и так, как нам удобнее.

Рассмотрим текст нашего do-файла (назовем его *make.do*). Содержимое файла *make.do* можно разделить на четыре части:

- компиляция файлов проекта;
- запуск симулятора;
- оформление окна **Wave**;
- запуск процесса моделирования.

Для компиляции проекта необходимо использовать команду ***vlog*** со списком имен всех файлов проекта:

```
vlog tb.v \
    Top_MIL_1553B.v \
    Receiver.v \
    Transmitter.v \
    RT_control.v \
    device3.v \
    device5.v \
    mem_dev3.v \
    mem_dev5.v
```

Список можно представить одной строкой, а также переносить на следующую строку с помощью знака «\».

Следующей строкой мы запускаем симулятор:

```
vsim work.tb
```

Здесь work — название библиотеки, которая по умолчанию указывается при создании проекта, а tb — имя нашего тестбенч-файла.

Следующим шагом будет оформление окна симуляции **Wave**. Читатель должен помнить, что отладка HDL-проекта пройдет настолько быстро, насколько удобно и функционально будет оформлено окно **Wave**.

Для начала пропишем пути к модулям проекта:

```
set dir_receiver /tb/Top_MIL_1553B/Receiver
set dir_transmitter /tb/Top_MIL_1553B/Transmitter
set dir_rt_control /tb/Top_MIL_1553B/RT_control
set dir_device3 /tb/Top_MIL_1553B/RT_control/device3
set dir_device5 /tb/Top_MIL_1553B/RT_control/device5
```

Здесь мы просто назначаем длинным путям отдельное имя, которое неоднократно будем использовать далее при добавлении отдельных сигналов.

Для добавления сигналов в окно **Wave** служит команда ***add wave*** с различными атрибутами. Например, чтобы выделять некоторые сигналы в группы, используется атрибут -group <имя группы>, причем если в имени группы присутствуют пробелы, то это имя необходимо указывать в фигурных скобках {}.

Добавим два глобальных тактовых сигнала и сигнал аппаратного сброса:

```
add wave -noupdate -format Logic -label clk /tb/clk
add wave -noupdate -format Logic -label reset /tb/reset
```

Следует указать, что если мы не станем использовать атрибут -label <имя сигнала>, то в окне **Wave** будет указан полностью путь до сигнала, что не совсем удобно.

Далее добавим сигналы, ответственные за основной канал драйвера протокола:

```
add wave -noupdate -group {1553B channel A} \
    -format Logic -label DI1A /tb/DI1A
add wave -noupdate -group {1553B channel A} \
    -format Logic -label DI0A /tb/DI0A
add wave -noupdate -group {1553B channel A} \
    -format Logic -label DO1A /tb/DO1A
add wave -noupdate -group {1553B channel A} \
    -format Logic -label DO0A /tb/DO0A
add wave -noupdate -group {1553B channel A} \
    -format Logic -label RX_STROB_A /tb/RX_STROB_A
add wave -noupdate -group {1553B channel A} \
    -format Logic -label TX_INHIBIT_A /tb/TX_INHIBIT_A
```

Можно заметить, что здесь сигналы указаны в составе общей группы с именем 1553B channel A. Аналогичным способом нужно сформировать сигналы для резервного канала, сигналы интерфейсов внутренней памяти и каждого из модулей проекта.

В качестве еще одного примера приведем добавление сигналов модуля RT_control.v:

```
add wave -noupdate -group RT_control \
    -format Logic -label clk $dir_rt_control/clk
add wave -noupdate -group RT_control \
    -format Logic -label rx_done $dir_rt_control/rx_done
add wave -noupdate -group RT_control \
     -format Logic -radix hexadecimal \
    -label rx_data $dir_rt_control/rx_data
…
add wave -noupdate -group RT_control \
    -format Logic -radix unsigned \
    -label addr_wr_dev5 $dir_rt_control/addr_wr_dev5
add wave -noupdate -group RT_control \
    -format Logic -label clk_wr_dev5 $dir_rt_control/clk_wr_dev5
add wave -noupdate -group RT_control \
    -format Logic -label we_dev5 $dir_rt_control/we_dev5
add wave -noupdate -group RT_control \
    -format Logic -label busy_dev5 $dir_rt_control/busy_dev5
```

Здесь в некоторых сигналах указан атрибут -radix, который позволяет указывать отображение сигнала в шестнадцатеричном (hexadecimal), десятичном беззнаковом (unsigned) и других форматах.

Наконец нам осталось запустить симуляцию, например на 500 мкс. Для этого добавим в наш do-файл команду:

```
run 500 us
```

Таким образом, только набрав в консоли текст **do make.do**, вы автоматически скомпилируете проект и запустите его на симуляцию с оформленным окном **Wave**.

Проанализировав результаты моделирования, то есть изучив полученные временные диаграммы и сообщения в консоли, разработчик проекта на ПЛИС может с максимальной уверенностью убедиться в работоспособности проекта. Если же в ходе разработки была допущена ошибка, то исправление ее не займет много времени.

На рис. 21 представлено окно **Wave** после запуска *make.do*. Здесь раскрыты две группы сигналов — основной канал протокола и модуль RT_control.v. На рис. 22 показан увеличенный фрагмент передачи пакета для субадреса 5.

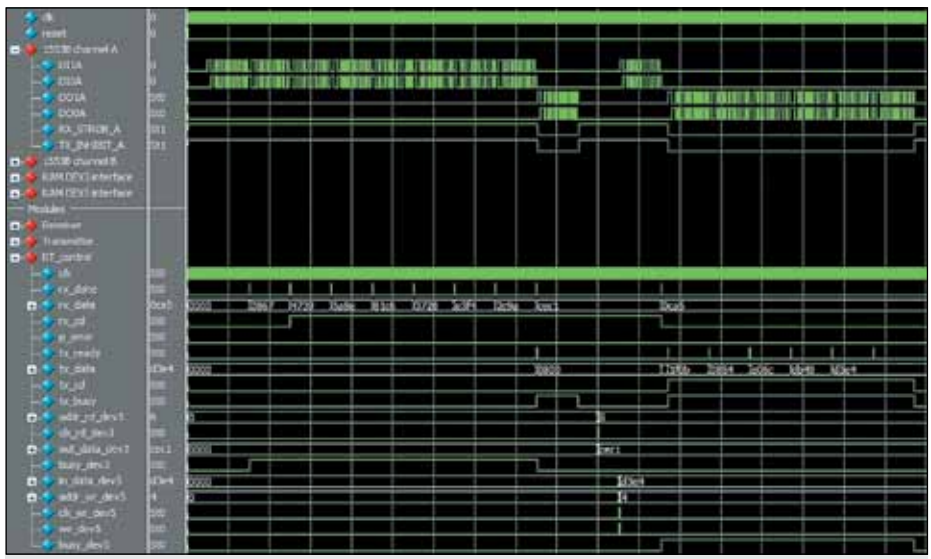

Рис. 21. Окно Wave в результате моделирования

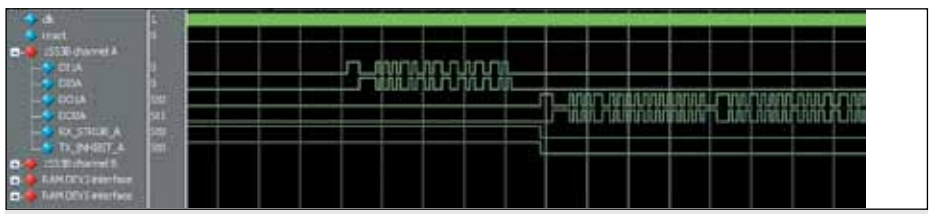

Рис. 22. Командное, ответное и информационные слова

Заключение

Описанный проект контроллера протокола MIL-STD-1553B автор реализовал на ПЛИС EP3C55F484. Фиттер САПР Quartus 9.0 Web Edition разместил проект на 333 логических элементах (LE). Из этого числа модуль передатчика занял 57 LE, а модуль приемника — 74 LE. Под RAM-память было выделено два блока памяти M9K. Как видим, все это не потребовало большого количества ресурсов. Автор для своего проекта использовал именно эту ПЛИС, с количеством более 55 тысяч LE, потому что в составе реального устройства, помимо драйвера протокола, было еще много другой периферии разного уровня сложности и требующей различных вычислений. Очевидно, что под реализацию только контроллера протокола можно использовать менее сложную ПЛИС.

Читателю, помимо временных диаграмм симулятора, будет полезно увидеть реальные осциллограммы пакетов протокола MIL-STD-1553B. На рис. 23 представлена осциллограмма начала пакета передачи информации от контроллера канала на оконечное устройство. Верхняя осциллограмма — сигнал между трансформатором и драйвером. Нижняя осциллограмма — сигнал между драйвером и ПЛИС. По синхросигналам можно отличить командное слово от информационного.

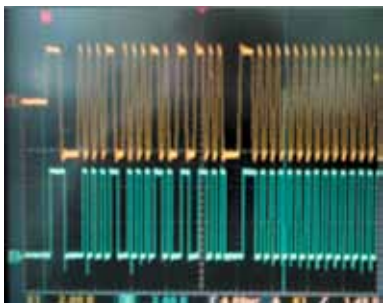

Рис. 23. Командное и информационное слова до драйвера и после него

На рис. 24 представлено то же командное слово, но в большем масштабе. На рис. 25 показана разность амплитуд сигналов в магистрали (до трансформатора) и после преобразования в трансформаторе. На рис. 26 видно, что по мере прохождения по магистрали уровень сигнала падает. Уровень сигнала ответного слова тут чуть выше, чем у командного и информационного слова. На рис. 27 приведен пакет передачи данных на оконечное устройство. Осциллограммы сняты между драйвером и ПЛИС.

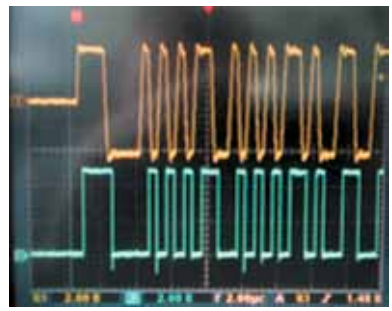

Рис. 24. Командное слово до драйвера и после него

Использование грамотно написанных тестбенч-модулей открывает перед разработчиком широкие возможности при проверке и отладке проекта. Такие модули просто необходимы при создании сложных проектов. ■

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.
6. Дайнеко Д. Разработка контроллера протокола MIL-STD-1553B на ПЛИС. Ч. 2 // Компоненты и технологии. 2014. № 1.
7. Дайнеко Д. Разработка контроллера протокола MIL-STD-1553B на ПЛИС. Ч. 3 // Компоненты и технологии. 2014. № 2.

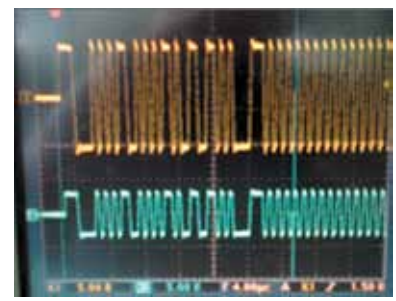

Рис. 25. Разница амплитуды сигнала на магистрали и после трансформатора

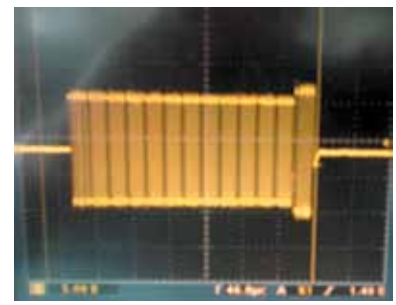

Рис. 26. Разница амплитуды сигнала принимаемых слов (КС, ИС) и передаваемого (ОС)

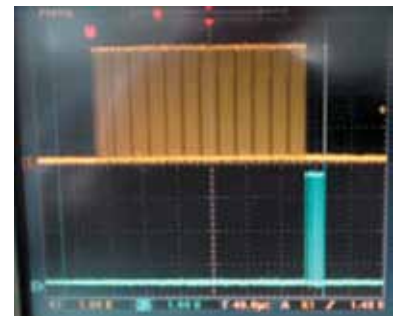

Рис. 27. Между драйвером и ПЛИС — командное слово, информационные слова и ответное слово