Тема 9. Передача данных между внешним устройством и FPGA на примере RS232

На ПЛИС возможно организовать эффективную обработку данных. Это может быть и вычисления специального назначения, обработка сигналов при приеме/передачи радио сигналов, видео потоков. Но в любом случае перед тем, как данные будут обработаны на FPGA, необходимо доставить эти данных «внутрь» чипа ПЛИС. Для этой цели существуют множество стандартов и протоколов передачи данных. Все эти стандарты и протоколы принято делить на так называемые уровни, например, широко известна сетевая модель передачи данных OSI (open system interconnect) http://ru.wikipedia.org/wiki/Сетевая_модель_OSI . Вкратце уровни передачи данных делятся на физический и логический. Логический уровень в свою очередь может включать в себя множество подуровней, но нам это сейчас неинтересно. Физический уровень стандартизует набор физических линий данных (проводников или проводов), по которым распространяются электрические сигналы, и их электрические характеристики (уровени напряжения для кодирования логического 0 и 1, частоты передачи данных, электрическое сопротивление проводников и тд). Эти физические проводники посредством соединений на печатной плате соединяются напрямую с контактами чипа ПЛИС. Логический уровень (или протокол) стандартизует, как воспринимать изменения напряжения на физических проводниках. Например, один из проводников может служить для передачи тактового сигнала, а по другому проводнику могут последовательно передаваться байты данных бит за битом. Соответственно внутри ПЛИС необходимо реализовать так называемое IP (intellectual property) ядро, которое будет взаимодействовать с этими проводниками и воспринимать изменения сигналов на них в соответствии с протоколом передачи данных при приеме данных, а также генерировать изменения напряжений на проводниках при передаче. Приняв, байт данных в последовательной форме и записав его в сдвиговый регистр это ядро затем уже выдает это данное далее внутрь ПЛИС в параллельной. Как воспринимать этот байт данных (как начало команды, некоторого адреса или байта данных для обработки) стандартизует более высокий логический уровень модели передачи данных.

Существует множество стандартов интерфейсов передачи данных, различающихся как на физическом, так и на логическом уровнях. Не стремясь охватить и классифицировать их все, отметим, что интерфейсы бывают

- последовательные, байты передаются последовательно по одной линии бит за битом, пример – RS232, USB

- параллельные, когда несколько физических проводников объединяются в шину, по которой передается сразу N битов за один такт, пример – шина PCI.
- последовательно – параллельные, когда по каждой линии данные передаются последовательно, но таких линий много, и по каждой их них принимается свой байт, пример – PCI Express.

Также эти интерфейсы делят на синхронные и асинхронные. В составе синхронного интерфейса есть особый проводник для передачи тактового сигнала, по фронту которого осуществляется прием/передача очередного бита данных. Асинхронные стандарты предполагают, что взаимодействующие агенты (приемник и передатчик) заранее договорились о скорости передачи данных и «ловят» биты данных в строго определенные моменты времени.

Все эти интерфейсы различаются характеристиками (скорость передачи данных, латентность, надежность передачи данных) а также сложностью проектирования и использования. Например, шина PCI Express http://ru.wikipedia.org/wiki/PCI_Express, которая сейчас является стандартом передачи данных между компонентами компьютера, может обеспечивать крайне высокие скорости передачи данных (от 250 МБайт/с до 32 Гбайт/с) при низкой латентности и высокой надежности передачи. Это достигается благодаря крайне высокой частоте передачи данных от 2.5 ГГц до 8 ГГц, а также использованию специальных приемом для зашиты от помех и сложного протокола. Интерфейс PCI Express доступен и в ПЛИС. Существуют специальные IP ядра, обеспечивающие весь протокол передачи данных, но эти ядра довольно сложны (изучение как их использовать – отдельная тема) и не годятся для обучения базовым вещам. Хотя, надо отметить, что этот интерфейс широко применяется при организации вычислений на FPGA в суперкомпьютерной области.

Мы же в методологических целях займемся изучением протокола передачи данных RS232 и сами создадим интерфейсное IP ядро. Этот же интерфейс и будет использован для передачи данных между ПЛИС и персональным компьютером для организации вычислений на ПЛИС.

Интерфейс RS232 предназначен для соединения двух и только двух агентов по принципу точка – точка. В простейшем (и самом распространенном) случае интерфейс RS232 состоит всего из 3 линий проводников: линия земли и двух линий данных: линия на прием RX и линия на передачу TX. При этом с одной стороны интерфейса линия на прием называется RX, но с другой стороны эта же линия является передающей и называется TX. С другой линией дела обстоят точно также. Протокол передачи данных для обеих линий одинаков, так что имеет смысл рассмотреть передачу данных только по одной линии. Так как для передачи данных в одном направлении логически используется только 1 линия (линия земли не участвует в передачи данных и служит только для электрического замыкания контура), то это протокол относят к классу асинхронных последовательных протоколов передачи данных. На лини данных определены всего два состояния – логической единицы и логического нуля. Реальные значения напряжений, соответствующих этим состояниям могут быть разные. Важнейшая характеристика интерфейса RS232 является скорость передачи данных в одном направлении. Измеряется она в бодах в секунду (baud/sec) и равна общему количеству переданных битов в секунду. Я говорю общему,

потому что не все биты, передаваемые по линии, являются битами данных, есть и служебные биты. Типичная скорость – это 115200 бод в секунду, что примерно равно 11 кбит/с. На самом деле скорость эта может изменяться в широких пределах от 9500 бод (и даже меньше) до 12 мбод. Рассмотрим протокол приема одного байта (рис 1). Передача может быть рассмотрена аналогично.

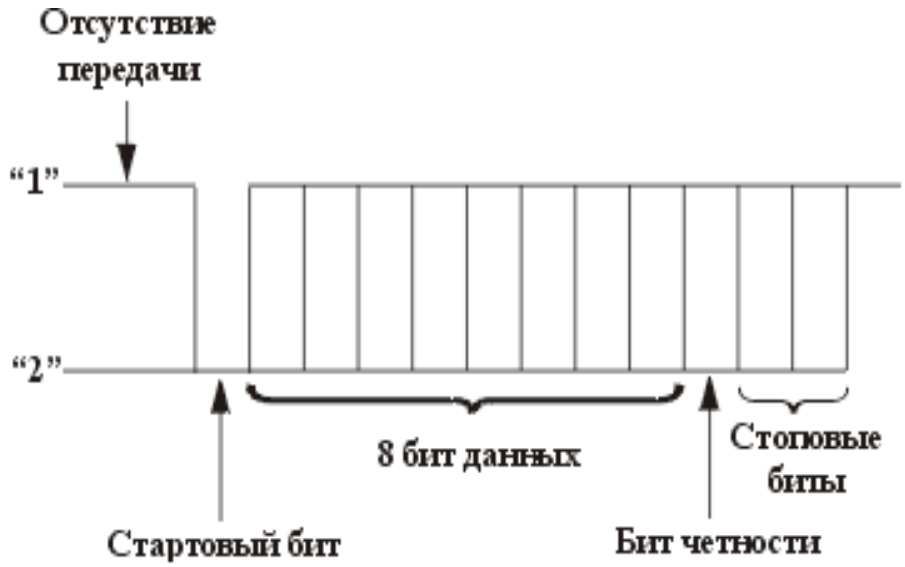


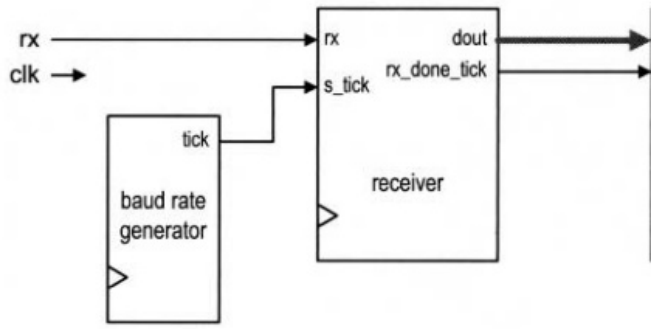
Рис 1. Протокол передачи одного байта по интерфейсу RS232. “1” – это логическая 1, “2” – логический 0.

В отсутствии передачи на линии поддерживается состояние логической 1. В этом состоянии приемник постоянно проверяет значение уровня на линии. Как только приемник фиксирует на линии логический 0 (говорят, что пришел стартовый бит), считается, что начинается передача байта данных. Стартовый бит является служебным битом и не является частью принимаемого байта. Это сделано для того, чтобы исключить ситуации, когда первый переданный бит данных будет равен 1 – в этом случае приемник пропустит этот бит (будет считать, что передача отсутствует). Таким образом, стартовый бит всегда равен 0 и определяется начало передачи данных. Приемник знает скорость передачи, а значит и длительность передачи одного бита. Он ожидает полтора таких периода, проверяет значение на линии и записывает его в сдвиговый регистр. Ожидание длится полтора периода, чтобы проверить значение на линии в момент, соответствующей передачи середины первого бита данных, чтобы гарантированно попасть в момент, когда значение на линии не меняется. Далее приемник последовательно ожидает по одному периоду и каждый раз сохраняет значение нового бита в сдвиговым регистре. После приема 8 бит, считается, что байт принят, и он выдается внутрь схемы ПЛИС в параллельной форме. Однако, опционально может передаваться еще и бит четности – свертка по модулю 2 передаваемого байта, вычисленная на стороне передатчика. Тогда приемник тоже может вычислить значение на своей стороне и сравнить с принятым значением бита четности. Если эти

значения совпадают, то считается, что байт принят без искажений, и он передается на обработку. В случае несовпадения считается, что один из битов был принят неправильно, и байт отбрасывается. Что делать в этом случае (переспрашивать байт или нет) – решается на более высоком уровне протокола передачи, который мы не рассматриваем. После приема бит четности идет всегда равный нулю стоповый бит, который и завершает транзакцию на линии. Надо сказать, что количество бит данных (7, 8 или 9), наличие бита четности и количество стоповых битов (1 или 2) – это опциональные параметры, но они должны быть известны переда началом передачи данных и совпадать на обоих концах линии. Обычно их настраивает оператор (или программа) при начале передачи.

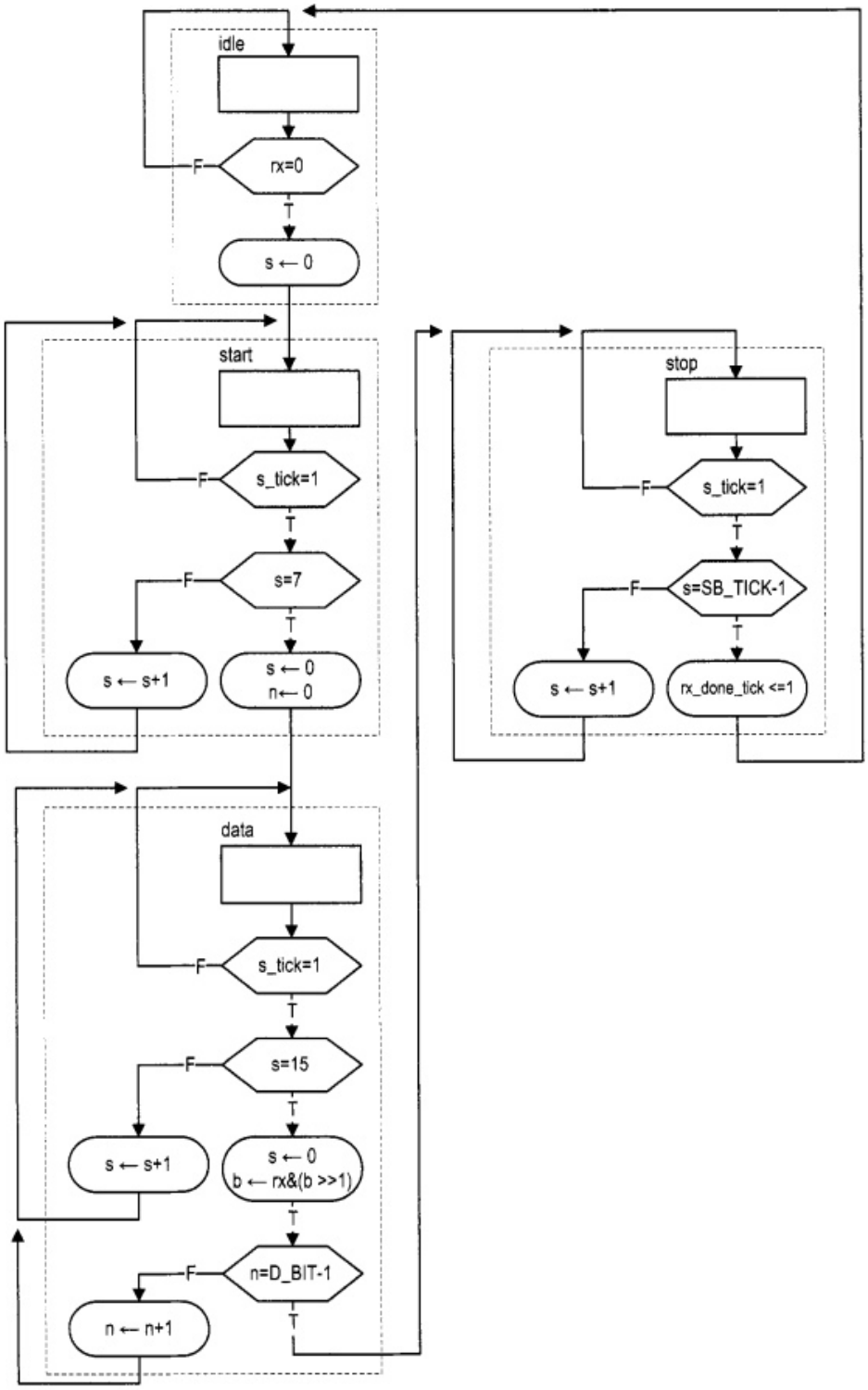
Мы начнем с реализации схемы приемника, которая имеет очень простой интерфейс:

- rx – вход, сигнал RX – последовательная линия данных RS232
- clk – вход, тактовый сигнал 100 MHz
- dout – 8 байтовая выходная шина данных – принятый байт
- rx_done_tick – выход, определяющий валидные даны на dout – однотактовый импульс



Во-первых, нам необходимо подстроить нашу схему под частоту передачи на RS232 – baud rate. Идея тут такая: мы делим период передачи одного бита на 16 равных частей – для того, чтобы точнее попадать в центр очередного бита. Для этого нам необходимо синтезировать частоту в 16 раз большую чем скорость передачи baud rate. При частоте baud rate 115200 бод/с, нам необходимо сгенерировать сигнал с 115200*16 = 1843200 периодами в секунду. У нас есть системная частота 100 МГц, и тогда период требуемого сигнал будет равен 100*10^6/1843200 = 54 периода системной частоты. Такой сигнал легко сформировать с помощью счетчика по модулю 54. Также в соответствии с принципом построения синхронной схемы мы не будем создавать новый тактовый сигнал – все наши триггеры будет работать на той же системной частоте 100 МГц – вместо этого мы будем генерировать сигнал разрешения работы enable. Этот сигнал будет иметь вид одного однотактового импульса системной частоты каждые 54 такта. Для этой роли сгодится просто счетчик, считающий от 0 до 53 на частоте 100 МГц. Когда значение счетчика равно 53, выдается однотактовый импульс s_tick.

Этот сигнал s_tick поступает на вход цифрового автомата, реализующего описанную выше логику приема байта данных по RS232. Рассмотрим его:



Опишем этот автомат на словах. Настроечными параметрами являются количество бит данных DBIT и количество стоповых бит (1, 1,5 или 2) в тиках s_tick (16, 24, 32) – SB_TICK. Проверка четности не производится. Сигнал s_tick – это выход схемы baud rate generator с периодом в 16 раз меньшим чем длительность передачи одного бита. Автомат имеет 4 состояния: idle, start, data и stop. В состоянии idle действий никаких не производится, и автомат просто ждет 0 на линии rx. Если на rx зафиксирован 0, то автомат переходит в состояние start, ждет в нем прихода 8 импульсов s_tick (что соответствует середине стартового бита) и переходит в состояние data. Отметим, что пока s_tick = 0, то автомат (и все сигналы) сохраняет свое состояние. Схема содержит два дополнительных счетчика, представленных регистрами s и n. Счетчик s увеличивается на 1 при приходе каждого импульса s_tick и считает до 7 в состоянии start, до 15 в состоянии data и до значения SB_TICK в состоянии stop. Регистр n содержит число принятых бит данных в состоянии data. Принятые биты данных записываются в сдвиговый регистр b. Также схема содержит выходной сигнал rx_done_tick, который равен 1 в течение одного такта системной частоты после окончания приема данных в регистр b. По этому сигналу необходимо считывать значение принятого байта с выходной шины dout.

```
-- UART RX
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_rx is
  generic(
    DBIT: integer:=8;     -- # data bits
    SB_TICK: integer:=16  -- # ticks for stop bits
  );
  port(
    clk, reset: in std_logic;
    rx: in std_logic;
    s_tick: in std_logic;
    rx_done_tick: out std_logic;
    dout: out std_logic_vector(7 downto 0)
  );
end uart_rx ;

architecture arch of uart_rx is
  type state_type is (idle, start, data, stop);
  signal state_reg, state_next: state_type;
  signal s_reg, s_next: unsigned(3 downto 0);
  signal n_reg, n_next: unsigned(2 downto 0);
  signal b_reg, b_next: std_logic_vector(7 downto 0);
begin
  -- FSMD state & data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      s_reg <= (others=>'0');
      n_reg <= (others=>'0');
      b_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
```

```
      b_reg <= b_next;
    end if;
  end process;
  -- next-state logic & data path functional units/routing
  process(state_reg,s_reg,n_reg,b_reg,s_tick,rx)
  begin
    state_next <= state_reg;
    s_next <= s_reg;
    n_next <= n_reg;
    b_next <= b_reg;
    rx_done_tick <='0';
    case state_reg is
      when idle =>
        if rx='0' then
          state_next <= start;
          s_next <= (others=>'0');
        end if;
      when start =>
        if (s_tick = '1') then
          if s_reg=7 then
            state_next <= data;
            s_next <= (others=>'0');
            n_next <= (others=>'0');
          else
            s_next <= s_reg + 1;
          end if;
        end if;
      when data =>
        if (s_tick = '1') then
          if s_reg=15 then
            s_next <= (others=>'0');
            b_next <= rx & b_reg(7 downto 1) ;
            if n_reg=(DBIT-1) then
              state_next <= stop ;
            else
              n_next <= n_reg + 1;
            end if;
          else
            s_next <= s_reg + 1;
          end if;
        end if;
      when stop =>
        if (s_tick = '1') then
          if s_reg=(SB_TICK-1) then
            state_next <= idle;
            rx_done_tick <='1';
          else
            s_next <= s_reg + 1;
          end if;
        end if;
    end case;
  end process;
  dout <= b_reg;
end arch;
```