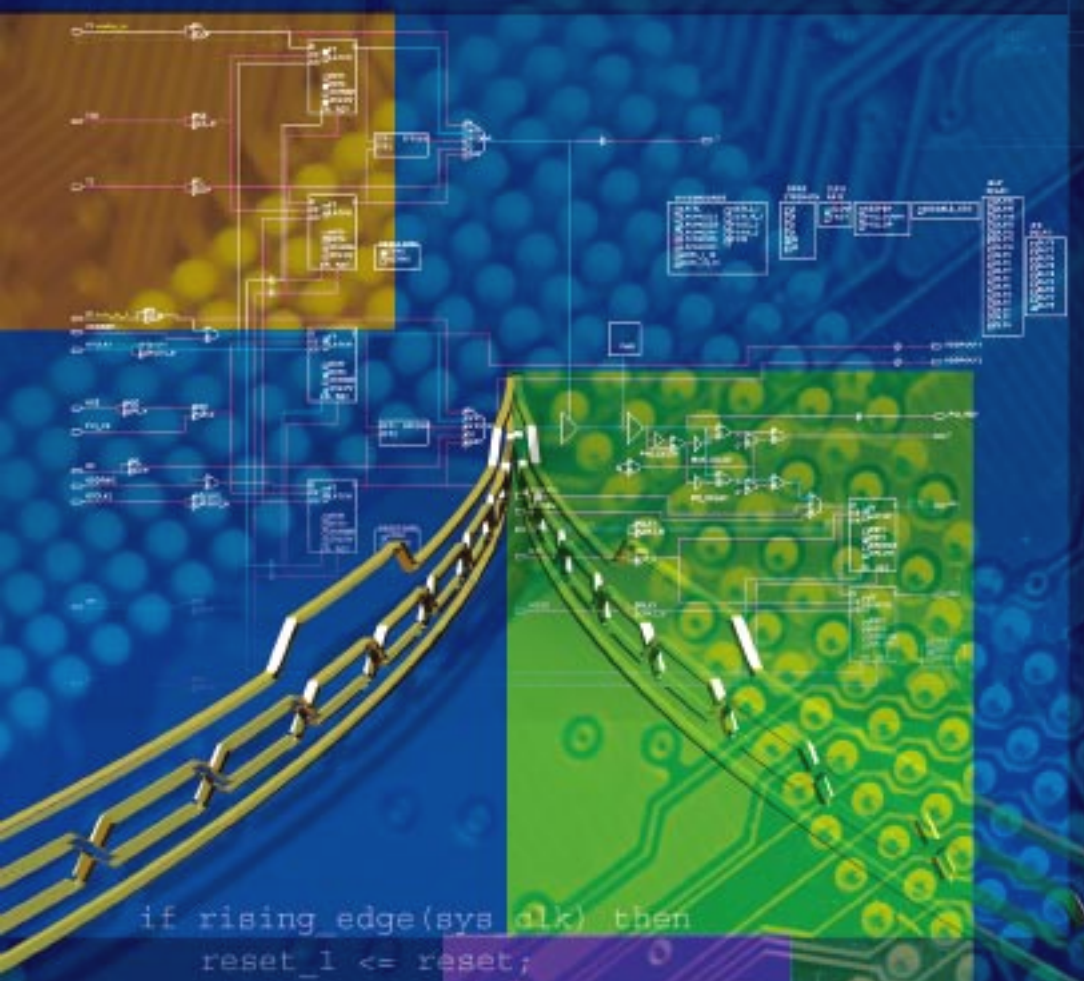
FPGA 时序收敛

一流设计让您高枕无忧。



作者：Nelson Lau
思博伦通信公司，首席硬件工程师
nelson.lau@spirent.com

您编写的代码是不是虽然在仿真器中表现正常，但是在现场却断断续续出错？要不然就是有可能在您使用更高版本的工具链进行编译时，它开始出错。您检查自己的测试平台，并确认测试已经做到 100% 的完全覆盖，而且所有测试均未出现任何差错，但是问题仍然顽疾难除。

虽然设计人员极其重视编码和仿真，但是他们对芯片在 FGPA 中的内部操作却知之甚少，这是情有可原的。因此，不正确的逻辑综合和时序问题（而非逻辑错误）成为大多数逻辑故障的根源。

但是，只要设计人员措施得当，就能轻松编写出能够创建可预测、可靠逻辑的 FPGA 代码。

在 FPGA 设计过程中，需要在编译阶段进行逻辑综合与相关时序收敛。而包括 I/O 单元结构、异步逻辑和时序约束等众多方面，都会对编译进程产生巨大影响，致使其每一轮都会在工具链中产生不同的结果。为了更好、更快地完成时序收敛，我们来进一步探讨如何消除这些差异。

I/O 单元结构

所有 FPGA 都具有可实现高度定制的 I/O 引脚。定制会影响到时序、驱动强度、终端以及许多其它方面。如果您未明确定义 I/O 单元结构，则您的工具链往往会采用您预期或者不希望采用的默认结构。如下 VHDL 代码的目的是采用“sda: inout std_logic;”声明创建一个称为 sda 的双向 I/O 缓冲器。

```
tri_state_proc : PROCESS (sys_clk)
BEGIN
      if rising_edge(sys_clk) then
       if (enable_in = '1') then
        sda <= data_in;
       else
          data_out <= sda;
          sda <= 'Z';
       end if;
      end if;
END PROCESS tri_state_proc;
```

当综合工具发现这组代码时，其中缺乏如何实施双向缓冲器的明确指示。因此，工具会做出最合理的猜测。

实现上述任务的一种方法是，在 FPGA 的 I/O 环上采用双向缓冲器（事实上，这是一种理想的实施方式）。另一种选择是采用三态输出缓冲器和输入缓冲器，二者都在查询表 (LUT) 逻辑中实施。最后一种可行方法是，在 I/O 环上采用三态输出缓冲器，同时在 LUT 中采用输入缓冲器，这是大多数综合器选用的方法。这三种方法都可以生成有效逻辑，但是后两种实施方式会在I/O 引脚与 LUT 之间传输信号时产生更长的路由延迟。此外，它们还需要附加的时序约束，以确保时序收敛。FPGA 编辑器清晰表明：在图 1 中，我们的双向 I/O 有一部分散布在 I/O 缓冲器之外。

教训是切记不要让综合工具猜测如何实施代码的关键部分。即使综合后的逻辑碰巧达到您的预期，在综合工具进入新版本时情况也有可能发生改变。应当明确定义您的 I/O 逻辑和所有关键逻辑。以下 VHDL 代码显示了如何采用 Xilinx® IOBUF 原语对 I/O 缓冲器进行隐含定义。另外需要注意的是，采用相似方式明确定义缓冲器的所有电气特性。

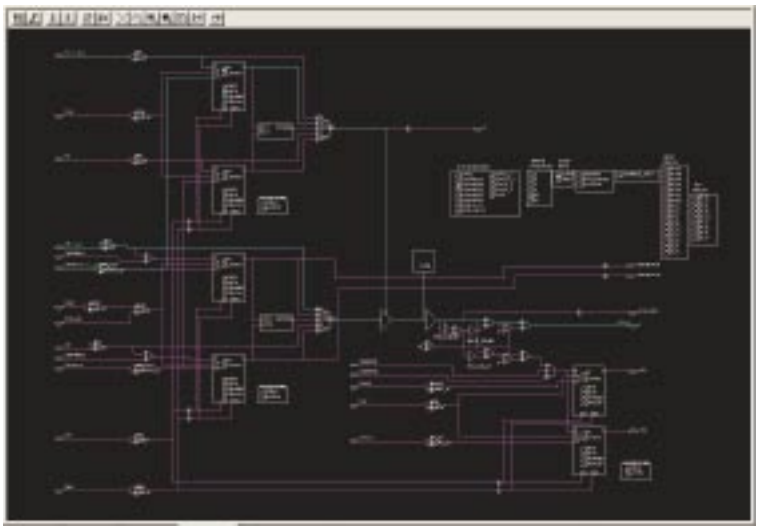


图 1 - FPGA 编辑器视图显示了部分双向 I/O 散布在 I/O 缓冲器之外

```
sda_buff: IOBUF
generic map (IOSTANDARD =>
"LVCMOS25",
IFD_DELAY_VALUE => "0", DRIVE =>
12,
SLEW => "SLOW")
port map(o=> data_out, io=> sda,
i=> data_in, t=> enable_in);
```

在图 2 中，FPGA 编辑器明确显示，我们已完全在 I/O 缓冲器内部实施了双向 I/O。

异步逻辑的劣势

异步代码会产生难以约束、仿真及调试的逻辑。异步逻辑往往产生间歇性错误，而且这些错误几乎无法重现。另外，无法生成用于检测异步逻辑所导致的错误的测试平台。

虽然异步逻辑看起来可能容易检测，但是，事实上它经常不经检测；因此，设计人员必须小心异步逻辑在设计中隐藏的许多方面。所有钟控逻辑都需要一个最短建立与保持时间，而且这一点同样适用于触发器的复位输入。以下代码采用异步复位。在此无法为了满足触发器的建立与保持时间需求而应用时序约束。

```
data_proc : PROCESS (sys_clk,reset)
BEGIN
      if (reset = '1') then
        data_in <= '0';
      elsif rising_edge(sys_clk) then
        data_in <= serial_in;
      end if;
END PROCESS data_proc;
```

下列代码采用同步复位。但是，大多数系统的复位信号都可能是按键开关，或是与系统时钟无关的其它信号源。尽管复位信号大部分情况是静态

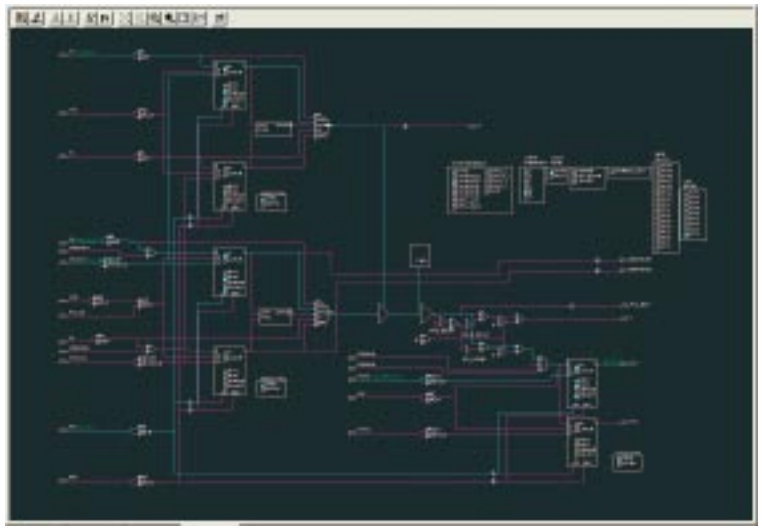


图 2 - 用一个VHDL代码转换明确定义 I/O 逻辑和关键逻辑，我们已完全在 I/O 缓冲器内部实施了双向 I/O

的，而且长期处于断言或解除断言状态，不过其水平仍然会有所变化。相当于系统时钟上升沿，复位解除断言可以违反触发器的建立时间要求，而对此无法约束。

```
data_proc : PROCESS (sys_clk)
BEGIN
      if rising_edge(sys_clk) then
       if (reset = '1') then
        data_in <= '0';
       else
        data_in <= serial_in;
       end if;
      end if;
END PROCESS data_proc;
```

只要我们明白无法直接将异步信号馈送到我们的同步逻辑中，就很容易解决这个问题。以下代码创建一个称为 sys_reset 的新复位信号，其已经与我们的系统时钟 sys_clk 同步化。在异步逻辑采样时会产生亚稳定性问题。我们可以采用与阶梯的前几级进行了‘与’运算的梯形采样降低此问题的发生几率。

```
data_proc : PROCESS (sys_clk)
BEGIN
      if rising_edge(sys_clk) then
       reset_1 <= reset;
       reset_2 <= reset_1 and reset;
       sys_reset <= reset_2 and reset_1
       and reset;
      end if;

      if rising_edge(sys_clk) then
       if (sys_reset = '1') then
        data_in <= '0';
       else
        data_in <= serial_in;
       end if;
      end if;
END PROCESS data_proc;
```

至此，假定您已经慎重实现了所有逻辑的同步化。不过，如果您不小心，则您的逻辑很容易与系统时钟脱节。切勿让您的工具链使用系统时钟所用的本地布线资源。那样做的话您就无法约束自己的逻辑。切记要明确定义所有的重要逻辑。

以下 VHDL 代码采用赛灵思 BUFG 原语强制 sys_clk 进入驱动低延迟网络 (low-skew net) 的专用高扇出缓冲器。

```
gclk1: BUFG port map (I => sys_clk,O
=> sys_clk_bufg);
data_proc : PROCESS (sys_clk_bufg)
BEGIN
      if rising_edge(sys_clk_bufg) then
       reset_1 <= reset;
       reset_2 <= reset_1 and reset;
       sys_reset <= reset_2 and reset_1
       and reset;
      end if;
      if rising_edge(sys_clk_bufg) then
       if (sys_reset = '1') then
        data_in <= '0';
       else
        data_in <= serial_in;
       end if;
     end if;
END PROCESS data_proc;
```

某些设计采用单个主时钟的分割版本来处理反序列化数据。以下 VHDL 代码（nibble_proc进程）举例说明了按系统时钟频率的四分之一采集的数据。

```
data_proc : PROCESS (sys_clk_bufg)
BEGIN
      if rising_edge(sys_clk_bufg) then
       reset_1 <= reset;
       reset_2 <= reset_1 and reset;
       sys_reset <= reset_2 and reset_1
       and reset;
      end if;

      if rising_edge(sys_clk_bufg) then
       if (sys_reset = '1') then
       two_bit_counter <= "00";
       divide_by_4 <= '0';
       nibble_wide_data <= "0000";
       else
       two_bit_counter
       <= two_bit_counter + 1;
divide_by_4 <= two_bit_counter(0) and
two_bit_counter(1);
```

```
        nibble_wide_data(0)
        <= serial_in;
        nibble_wide_data(1)
        <= nibble_wide_data(0);
        nibble_wide_data(2)
        <= nibble_wide_data(1);
        nibble_wide_data(3)
        <= nibble_wide_data(2);
      end if;
    end if;
END PROCESS data_proc;
nibble_proc : PROCESS (divide_by_4)
BEGIN
      if rising_edge(divide_by_4) then
       if (sys_reset = '1') then
         nibble_data_in <= "0000";
      else
         nibble_data_in
         <= nibble_wide_data;
      end if;
     end if;
END PROCESS nibble_proc;
```

看起来好像一切都已经同步化，但是 nibble_proc 采用乘积项 divide_by_4 对来自时钟域sys_clk_bufg 的 nibble_wide_data 进行采样。由于路由延迟，divde_by_4 与 sys_clk_bufg 之间并无明确的相位关系。将 divide_by_4 转移到 BUFG 也于事无补，因为此进程会产生路由延迟。解决方法是将 nibble_proc 保持在 sys_clk_bufg 域，并且采用 divide_by_4 作为限定符，如下所示。

```
nibble_proc : PROCESS (sys_clk_bufg)
BEGIN
      if rising_edge(sys_clk_bufg) then
       if (sys_reset = '1') then
         nibble_data_in <= "0000";
      elsif (divide_by_4 = '1') then
         nibble_data_in
         <= nibble_wide_data;
      end if;
    end if;
END PROCESS nibble_proc
```

时序约束的重要性

如果您希望自己的逻辑正确运行，则必须采用正确的时序约束。如果您已经慎重确保代码全部同步且注册了全部 I/O，则这些步骤可以显著简化时序收敛。在采用上述代码并且假定系统时钟为100MHz 时，则只需四行代码就可以轻松完成时序约束文件，如下所示：

```
NET sys_clk_bufg TNM_NET =
sys_clk_bufg;
TIMESPEC TS_sys_clk_bufg = PERIOD
sys_clk_bufg 10 ns HIGH 50%;
OFFSET = IN 6 ns BEFORE sys_clk;
OFFSET = OUT 6 ns AFTER sys_clk;
```

请注意：赛灵思 FPGA 中 I/O 注册逻辑的建立与保持时间具有很高的固定性，在一个封装中切勿有太大更改。但是，我们仍然采用它们，主要用作可确保设计符合其系统参数的验证步骤。

三步简单操作

仅需遵循以下三步简单操作，设计人员即可轻松实施可靠的代码。

- 切勿让综合工具猜测您的预期。采用赛灵思原语对所有 I/O 引脚和关键逻辑进行明确定义。确保定义 I/O 引脚的电气特性；
- 确保逻辑 100% 同步，并且让所有逻辑参考主时钟域；
- 应用时序约束确保时序收敛。

只要遵循上述三个步骤，您就能够消除综合与时序导致的差异。扫除这两个主要障碍会让您获得具有 100% 可靠性的代码。

赛灵思高性能 40nm Virtex-6 和低功耗 45nm Spartan-6 FPGA 全面量产

2010 年 1 月全球可编程逻辑解决方案领导厂商赛灵思公司与全球领先的半导体代工厂商联华电子(UMC (NYSE: UMC; TSE: 2303))共同宣布，采用联华电子高性能 40nm 工艺的 Virtex®-6 FPGA，已经完全通过生产前的验证。这是双方工程团队为进一步提升良率、增强可靠性并缩短生产周期而努力合作的成果。Virtex-6 系列通过生产验证，意味着联华电子继 2009 年 3 月发布首批基于 40nm 工艺的器件后，正式将该工艺转入量产。

2010 年 3 月赛灵思公司和全球高级半导体技术领先者三星电子有限公司共同宣布，赛灵思 Spartan®-6 FPGA 系列已取得三星电子旗下晶圆代工厂三星代工（Samsung Foundry）的 45nm 工艺技术的全面生产认证。这种先进的工艺节点技术结合业界一流的 FPGA 设计，可实现低成本、低功耗、高性能的最佳平衡，从而使 Spartan-6 系列 FPGA 能够满足成本敏感型市场的各种应用需求。今天发布的消息标志着在三星代工制造的赛灵思 45nm Spartan-6 FPGA 系列已经能够立即实现量产供货。