4.3 VIVADO 时序约束

在 FPGA 的设计过程中，常常会因为各个部件之间存在的延迟导致整体功能上的偏差甚至无法实现，而恰当的时序约束能很好的解决这个问题。FPGA 的时序约束功能主要如下：

1.提高设计的工作频率。

对很多数字电路设计来说，提高工作频率非常重要，因为高工作频率意味着高处理能力。通过附加约束可以控制逻辑的综合、映射、布局和布线，以减小逻辑和布线延时，从而提高工作频率。

2.获得正确的时序分析报告。

几乎所有的 FPGA 设计平台都包含静态时序分析工具，利用这类工具可以获得映射或布局布线后的时序分析报告，从而对设计的性能做出评估。静态时序分析工具以约束作为判断时序是否满足设计要求的标准，因此要求设计者正确输入约束，以便静态时序分析工具输出正确的时序分析报告。

3.指定 FPGA/CPLD 引脚位置与电气标准。

FPGA/CPLD 的可编程特性使电路板设计加工和 FPGA/CPLD 设计可以同时进行，而不必等 FPGA/CPLD 引脚位置完全确定，从而节省了系统开发时间。这样，电路板加工完成后，设计者要根据电路板的走线对 FPGA/CPLD 加上引脚位置约束，使 FPGA/CPLD 与电路板正确连接。另外通过约束还可以指定 IO 引脚所支持的接口标准和其他电气特性。为了满足日新月异的通信发展，Xilinx 新型 FPGA/CPLD 可以通过 IO 引脚约束设置支持诸如 AGP、BLVDS、CTT、GTL、GTLP、HSTL、LDT、LVCMOS、LVDCI、LVDS、LVPECL、LVDSEXT、LVTTL、PCI、PCIX、SSTL、ULVDS 等丰富的 IO 接口标准。另外通过区域约束还能在 FPGA 上规划各个模块的实现区域，通过物理布局布线约束，完成模块化设计等。

Vivado 软件相比于 ISE 的一大转变就是约束文件，ISE 软件支持的是 UCF（User Constraints File），而 Vivado 软件转换到了 XDC（Xilinx Design Constrains）。XDC 主要基于 SDC（Synopsys Design Constraints）标准，另外集成了 Xilinx 的一些约束标准，可以说这一转变是 Xilinx 向业界标准的靠拢。

UCF 和 XDC 文件除了约束命令的差别外，还存在以下的差别：

（1）XDC 是顺序执行约束，每个约束指令都有优先级。越靠后的指令优先级越高，会覆盖之前对相同部件的约束，建议时序约束放在 I/O 约束之前。

（2）UCF 一般约束 nets 对象，而 XDC 约束类型为 pins，ports 和 cells 对象。

（3）UCF 约束默认不对异步时钟间路径进行时序分析，而 XDC 约束默认所有时钟是相关的，会分析所有路径，可以通过设置时钟组（set_clock_groups）取消时钟间的相关性。

4.3.1 时钟约束简介

在使用 VIVADO 进行 FPGA 设计时常涉及到的时钟约束如下：

(1)Clock

1）Create Clock：时钟约束必须最早创建，对 7 系列 FPGA 来说，端口进来的主时钟以及 GT 的输出 RXCLK/TXCLK 都必须由用户使用 create_clock 自主创建。Vivado 进行时序分析时，以主时钟的源端点作为延时计算起始点。

2）Create Generated Clock：衍生时钟是由设计内部产生，一般由时钟模块(MMCM

or PLL）或逻辑产生，并且对应有一个源时钟，源时钟可以是系统的主时钟或者另外一个衍生时钟。约束衍生时钟时，除了定义周期，占空比，还需要指明与源时钟的关系。

3)Set Clock Latency：FPGA 内部时钟通常由外部时钟源提供，经过 PLL/MMCM 生成内部所用时钟。从时钟源比如晶振或者上游芯片经过板级走线到 FPGA 的专用时钟管脚，这个过程必然有板级走线延迟；由 create_clock 定义的时钟端口到同步元件的时钟端口也必然有延迟。这两类延迟共同构成了时钟延迟，前者为时钟源延迟（Source Latency），后者为时钟网络延迟（Network Latency）。对于时钟网络延迟，Vivado 会自行分析计算；对于时钟源延迟，通过 set_clock_latency 来定义。

(2)Inputs

1)Set Input Delay：用于数据输入端口，调节数据输入与时钟输入到来的相互关系。当 FPGA 外部送入 FPGA 内部寄存器数据时，会有两个时钟 launch clock 和 latch clock，前者负责将数据从外部寄存器中送出，后者要在 setup 与 hold 都满足的条件下，将数据锁入 FPGA 内部寄存器。在这个过程中，如果 launch clock 已经将数据送出，并到达 FPGA 内部寄存器端口，而上一次的数据的 hold 时间还不足，就会冲掉前面的这个数据，导致 latch clock 锁存数据错误！方法就是用 set_input_delay 在数据到达时间（data_arrival）上加延时，让数据推迟到达，让 latch lock 有足够的时间（一般是 hold time）对数据锁存。

(3)Outputs

1)Set Output Delay：用于数据输出端口，调节数据输出与时钟输出的相互关系。当 FPGA 内部送出数据给外部器件时，有两个时钟 launch clock 与 latch clock，前者负责将数据从内部寄存器中送出，后者要在 setup 与 hold 都满足的条件下，将数据锁入外部寄存器。在这个过程中，就是要保证在时钟到来时数据准备好，并让时钟有足够的时间将数据打入外部寄存器中。但如果 latch clock 已经到来，由于板级延时等问题，数据未能如时到达，那 latch clock 没有足够的 setup 时间对数据采样，导致 clock 猜到的数据错误。解决方法就是用 set_output_delay 在数据到达时间（data_arrival）上加延时（负值），或者说对 latch clock 加延时（正值），表现为数据提前到达，或认为 latch clock 推迟到达，而其实质就是以时钟为参考，对数据进行的操作，让 latch clock 有足够的时间（一般为 setup time）对数据锁存。

4.3.2 添加时钟约束

1.时钟约束必须最早创建，对 7 系列 FPGA 来说，端口进来的主时钟以及 GT 的输出 RXCLK/TXCLK 都必须由用户使用 create_clock 自主创建。如图 4.3.1 所示，在 Flow Navigator 中选择 Synthesis > Synthesized Design > Edit Timing Constraints。

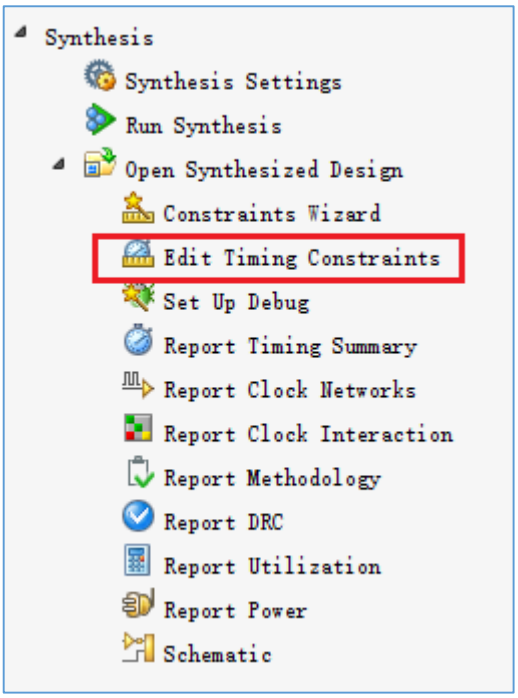

图 4.43 选择时钟约束

2.打开图 4.44 所示时序约束界面，开始进行时序约束。

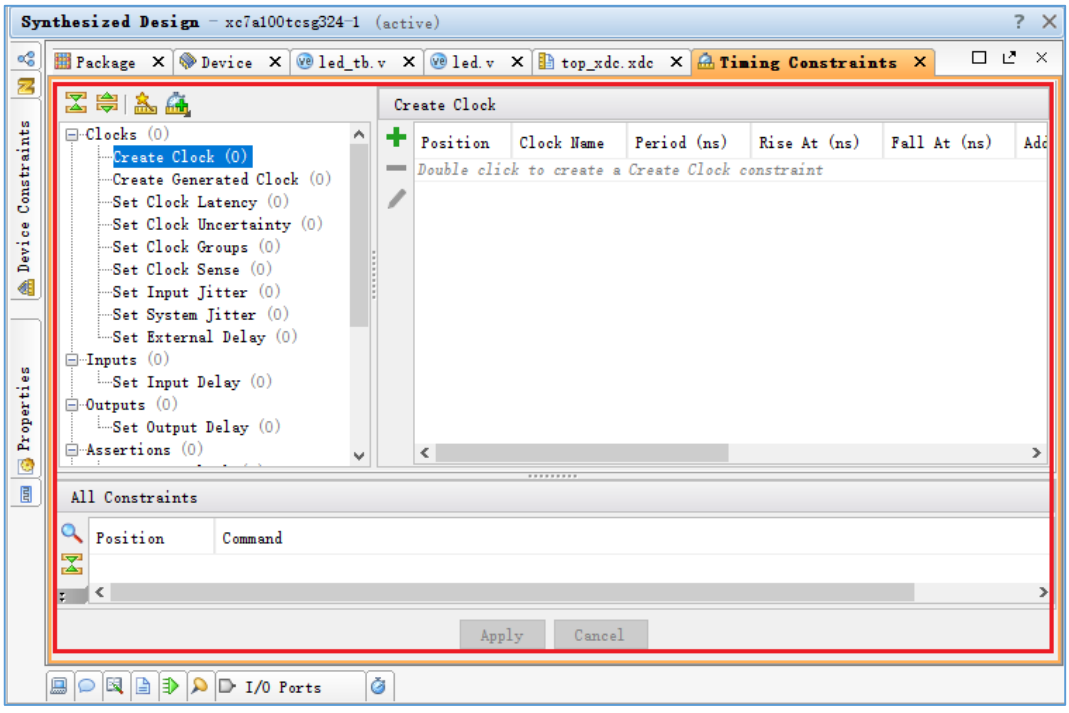

图 4.44 时钟约束

3.双击左边 Clock->Create Clock,进入图 4.45 所示 Create Clock 界面,在 Clock name 中输入时钟变量名，此处为 clk_pin。在 Source objects 中选择右边的按钮。

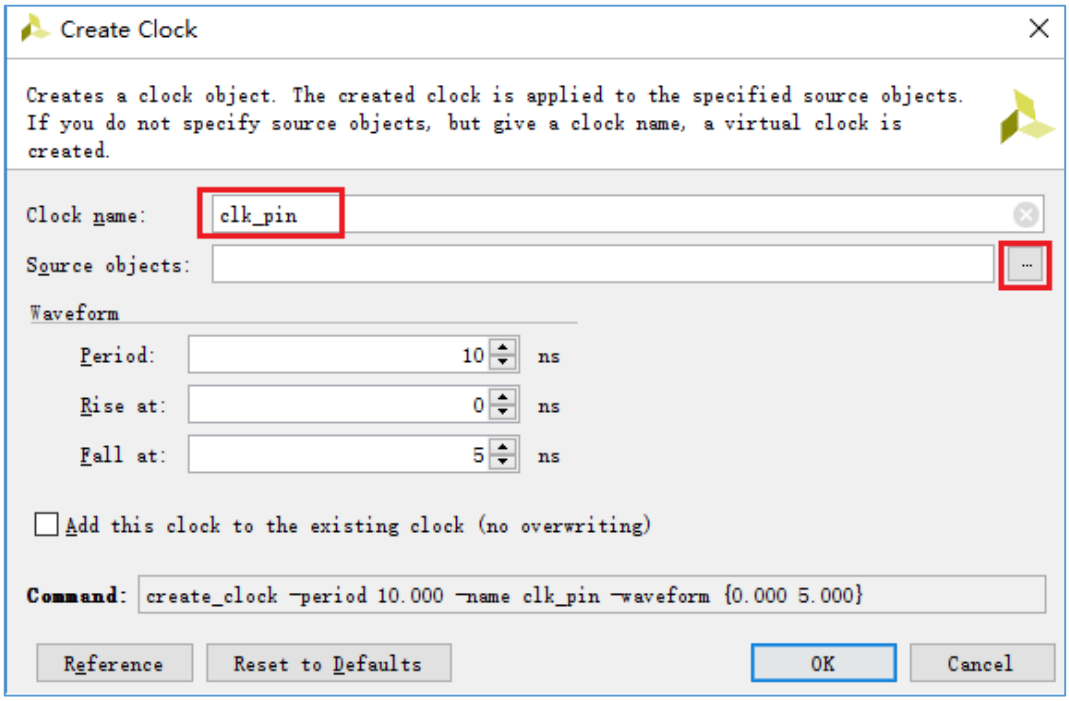

图 4.45 时钟变量名设置

4.在图 4.46 所示 Specify Clock Source Object 界面中，Find names of type 选项选择 I/O Ports 后点击 Find，并将查找到的 clk 选中移到选中框中。完成选择后点击 Set。此步骤将之前 I/O 约束中约束过的 clk 变量选中，为覆盖做好准备。

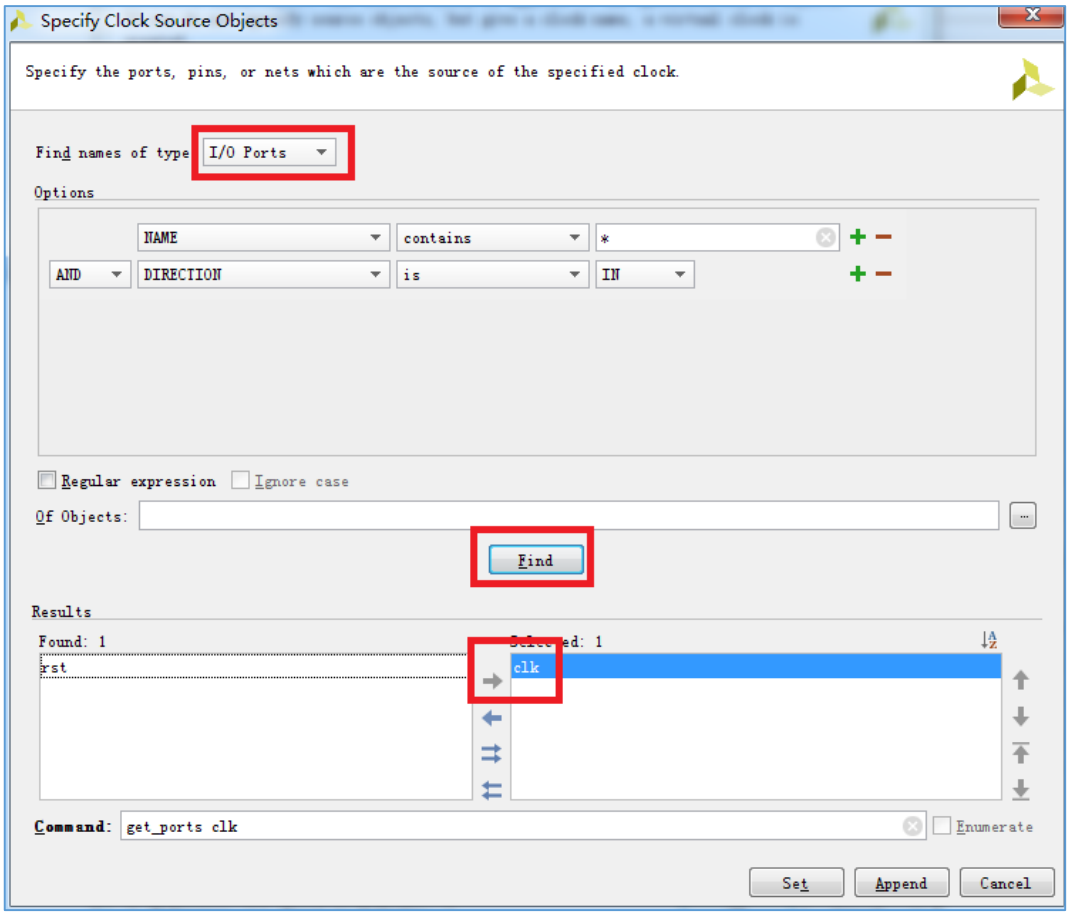

图 4.46 时钟源对象设置

5.在图 4.47 所示界面中将 Period 设置成 10ns、Rise 设置成 0ns、fall 设置成 5ns，并点击 ok。事实上，Xilinx 的 7 系列开发板的主频就是 100MHZ，即周期为 10ns，此步骤在一定程度上相当于对主频的降频。

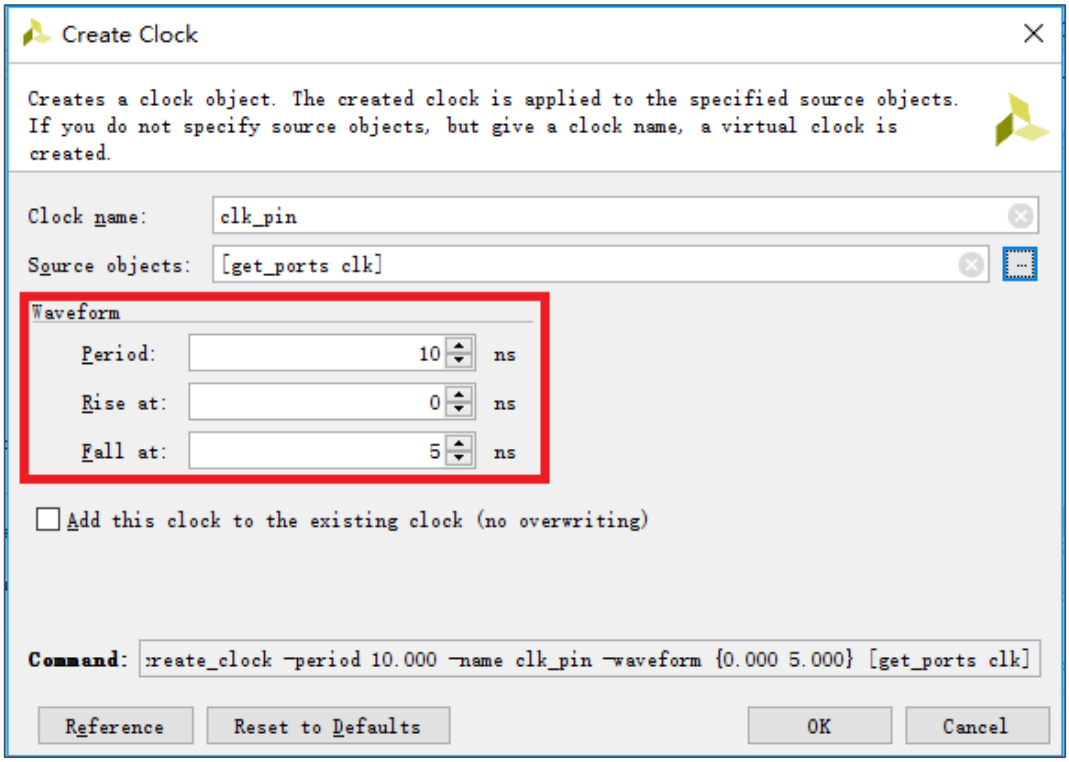

图 4.47 创建时钟

6.如图 4.48 所示，这时 Clock 已经创建完成。图 4.47 中的 command 已经添加到工程 xdc 文件中。

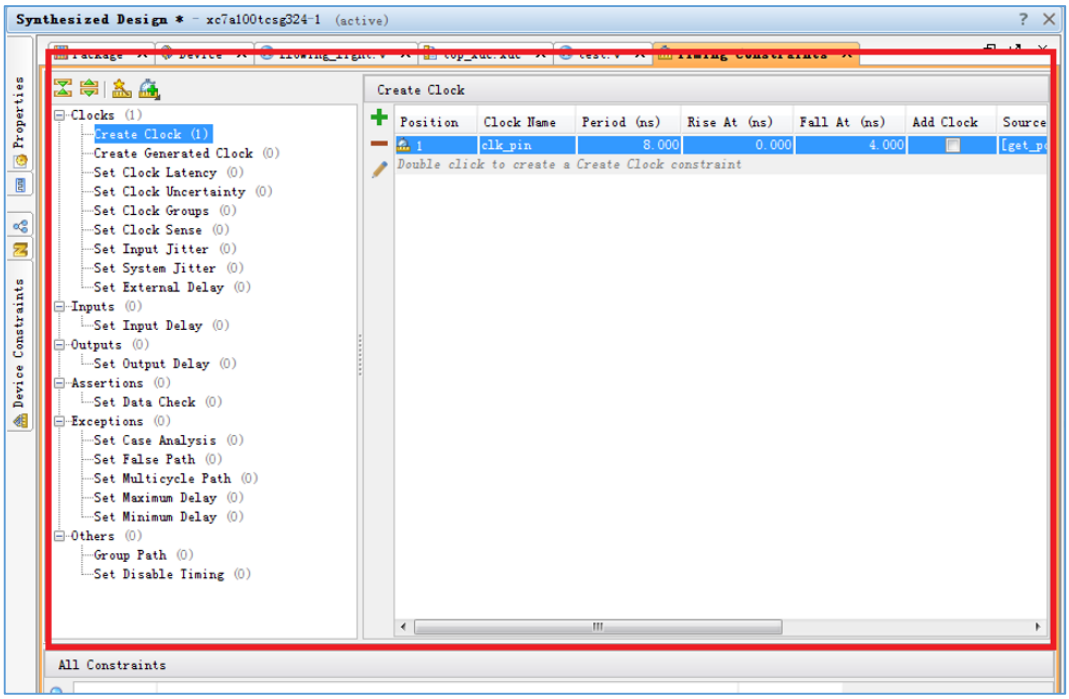

图 4.48 时钟创建完成

7.在一些简单的 FPGA 部件设计过程中，完成了上述的 create_clock 的设置基本上就能

使得时序满足要求。在一些比较复杂的设计中可能还会涉及到更多设置，其中使用较多的有 set_input_delay/set_output_delay。如果对 FPGA 的 I/O 不加任何约束，Vivado 会缺省认为时序要求为无穷大，不仅综合和实现时不会考虑 I/O 时序，而且在时序分析时也不会报出这些未约束的路径。接下来将设置 Input Setup Delay，如图 4.49 所示，双击左边 Input->Set Input Delay。

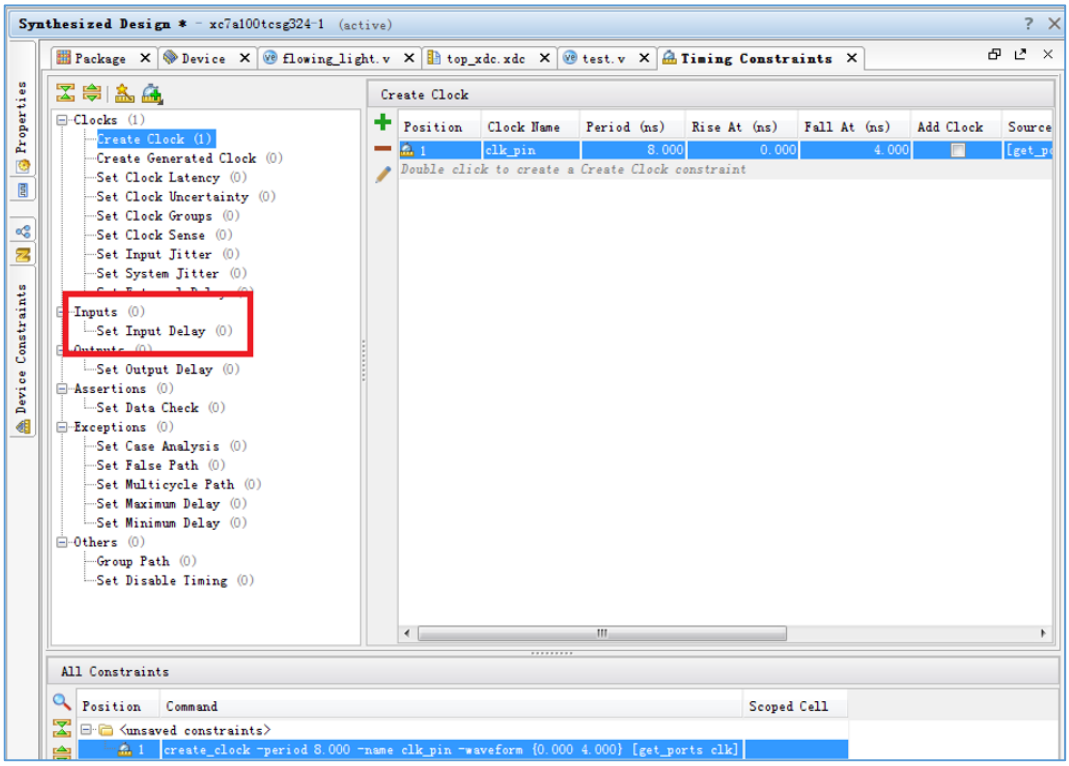

图 4.49 设置 Input Setup Delay

8.进入 Set Input Delay，按照图 4.50 配置，Clock 选择 clk_pin，指明了对 Objects 进行时序分析所用的时钟，Objects 选择 rst，这是想要设定 input 约束的端口名。Delay 选择需要的延迟时间，这里忽略之，设置成 0。如有必要，Min/Max 选择 max 时描述了用于 setup 分析的包含有板级走线和外部期间的延时；选择 min 时描述了用于 hold 分析的包含有板级走线和外部期间的延时。完成设置后点击 OK。设置完之后在 XDC 文件中会生成一句 set_input_delay -clock<clock_name><objects> -max <maxdelay> -min <mindelay>的约束代码。

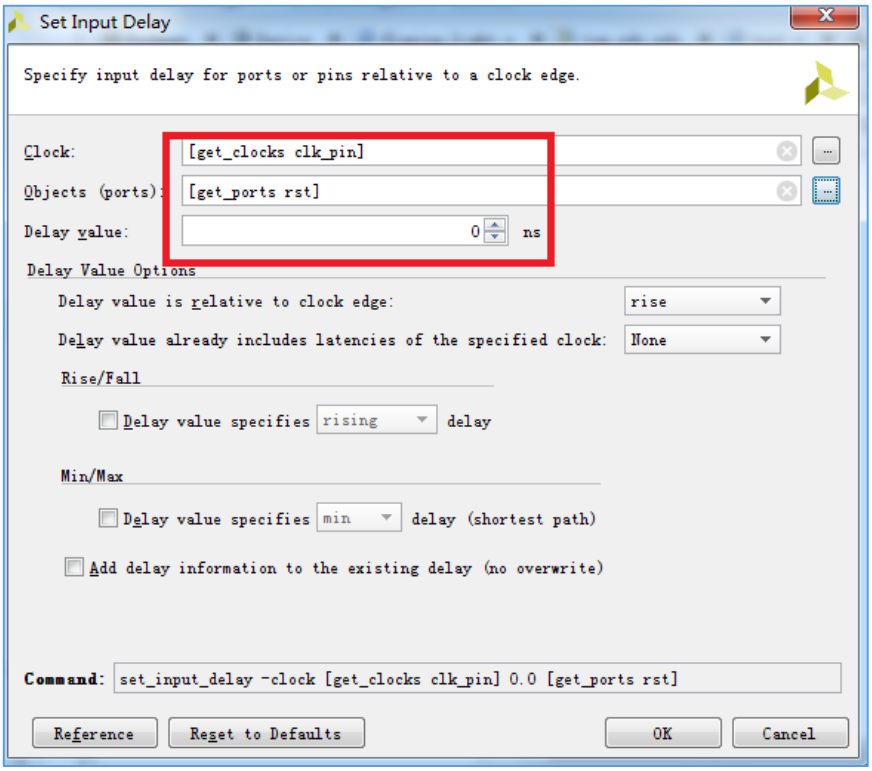

图 4.50Input Setup Delay 配置

9.接下来设置 Output Delay，其设置与 Input Delay 相似。如图 4.51 所示，双击左边 Output->Set Output Delay。Clock 选择 clk_pin、Objects 选择所有输出，Delay value 设置为 0ns，当然，有必要时也能设置 max/min delay。设置完之后会在 XDC 文件中生成一句 set_output_delay -clock<clock_name><objects> -max <maxdelay> -min <mindelay>的约束代码。

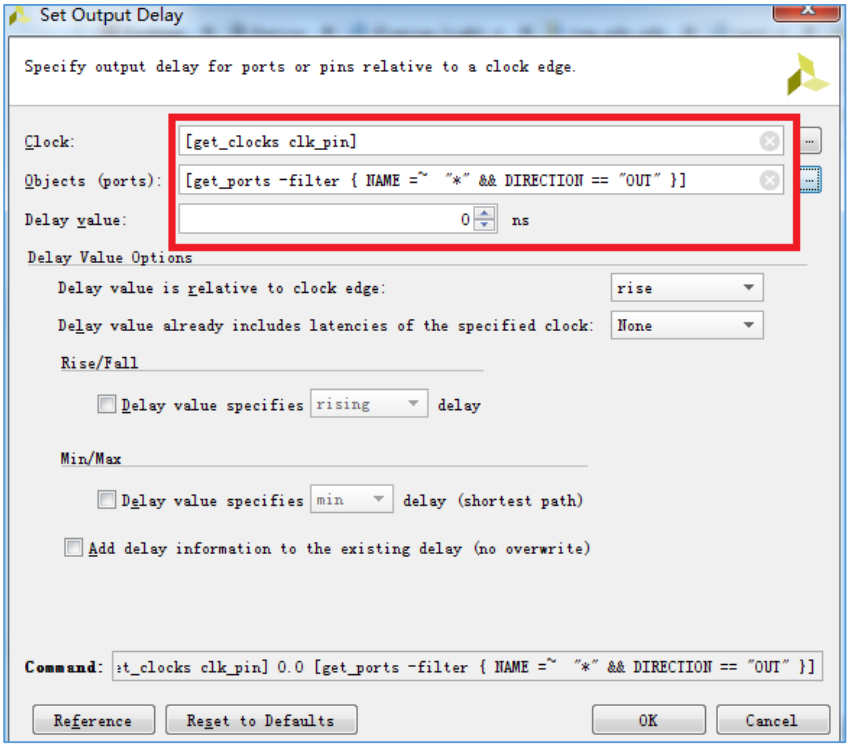

图 4.51Output Delay 配置

10.完成以上约束后选择 File->Save Constraints 将设置的约束保存，之后可以在 XDC 文件中看到如图 4.52 的约束结果。

```
create_clock -period 10.000 -name clk_pin -waveform {0.000 5.000} [get_ports clk]
set_input_delay -clock [get_clocks *] 0.000 [get_ports rst]
set_output_delay -clock [get_clocks *] 0.000 [get_ports -filter { NAME =~  "*" && DIRECTION == "OUT" }]
```

图 4.52 约束结果

4.3.3 Report Timing Summary 时序分析

1.完成时序约束之后，利用 Vivado 提供的时序分析手段进行分析，即使用 Report Timing Summary 进行分析。如图 4.53 所示，在 Flow Navigator 中选择 Synthesized Design->Report Timing Summary。并将 Options 标签里将 Path delay type 设置成 min_max。

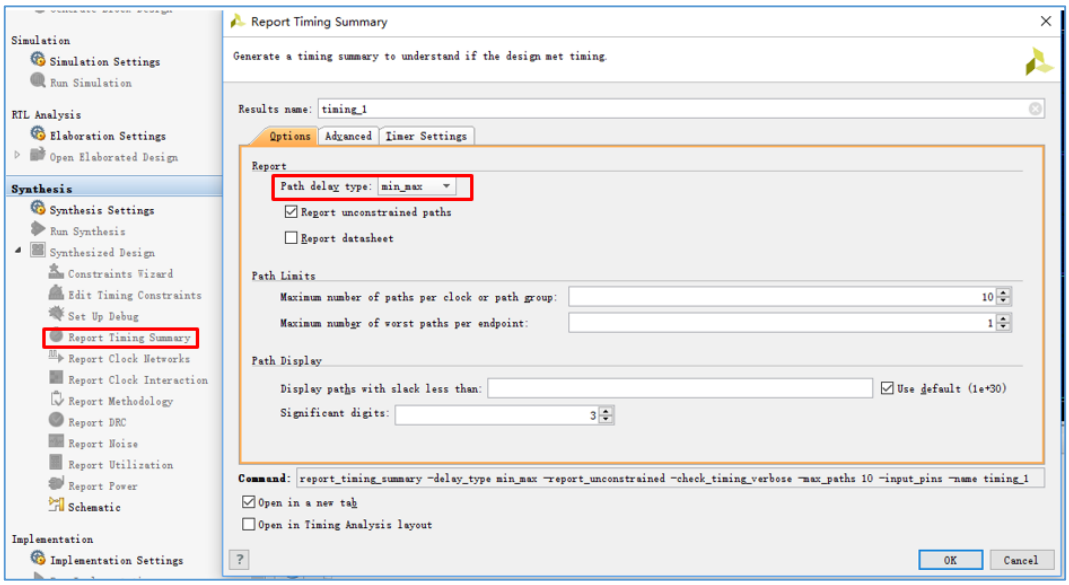

图 4.53 时序分析

reprt_timing_summary 实际上隐含了 report_timing、report_clocks、check_timing 以及部分的 report_clock_interaction 命令，所以最终看到的图 4.54 所示的报告中也包含了这几部分的内容。

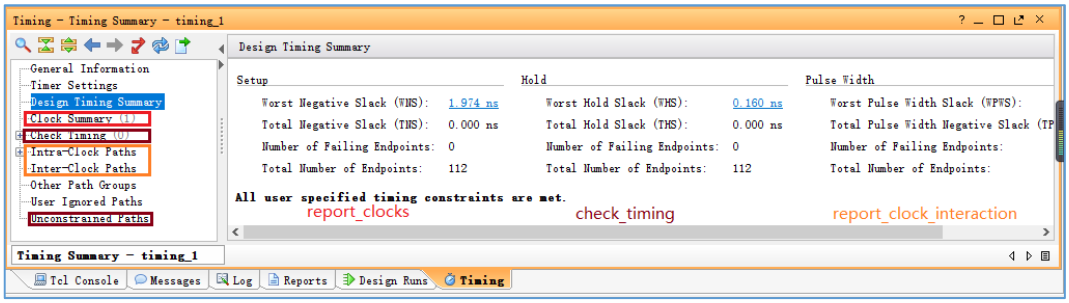

图 4.54 时序分析报告

Timing Summary 报告把路径按照时钟域分类，如图 4.55 所示，每个组别下缺省会报告 Setup、Hold 以及 Pulse Width 检查最差的各 10 条路径，还可以看到每条路径的具体延时报告，并支持与 Device View、Schematic View 等窗口之间的交互。

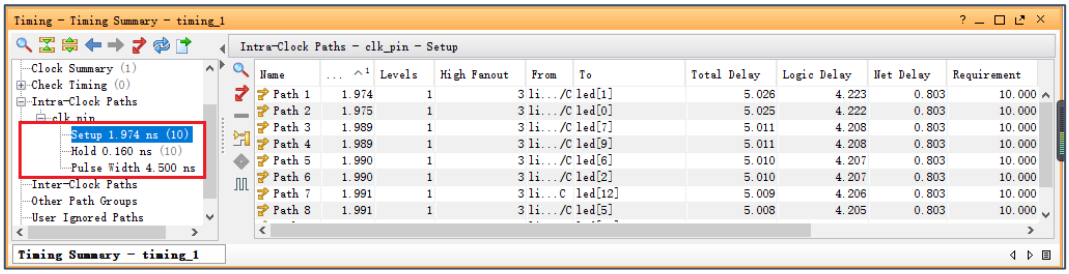

图 4.55 时序分析报告分类

如图 4.56 所示，每条路径具体的报告会分为 Summary、Source Clock Path、Data Path 和 Destination Clock Path 几部分，详细报告每部分的逻辑延时与连线延时。用户首先要关注的就是 Summary 中的几部分内容，发现问题后再根据具体情况来检查详细的延时数据。其中，Slack 显示路径是否有时序违例，Source 和 Destination 显示源驱动时钟和目的驱动时钟及其时钟频率， Requirement 显示这条路径的时序要求是多少，Data Path 显示数据路径上的延时，Logic Level 显示这条路径的逻辑级数，而 Clock Path Skew 和 Clock Uncertainty 则显示时钟路径上的不确定性。

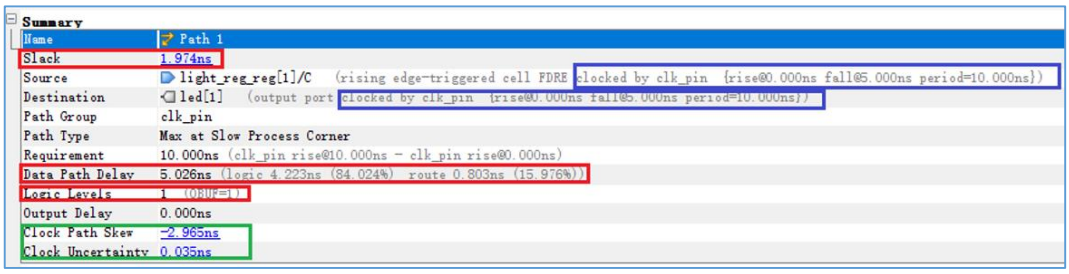

图 4.56 详细报告

将光标移动到有下划线的数据上然后单击，能得到相应数据的计算方式，如图 4.57。

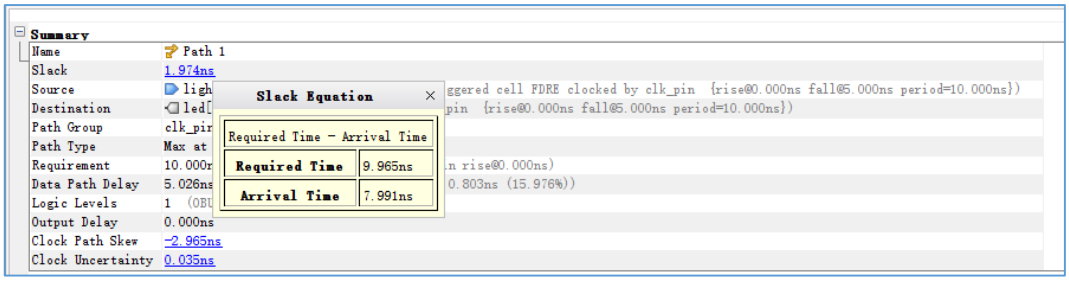

图 4.57 数据计算方式

通过 Summary 我们可以得到知道这一条 clk 时钟域内的路径的时钟周期，从 Slack 的值看出是否存在时序违例。而通过 Data Path Delay 和 Logic Levels 的值可以看出是否是因为连线延时比例过高或者是逻辑级数较高导致的数据链路延时较大，从而可以从改进布局、降低扇出或者是减少逻辑级数的优化方向。

2.为了使 ReportTiming Summary 的使用更加形象，这里使用了一个完整 CPU 的例子进行描述。

首先通过之前介绍的方法将 CPU 工程在 Vivado 中建立起一个完整的工程，再进行简单的时序约束，为了尽可能提高 CPU 的性能，在建立主时钟源时使用了一个理论上比较合适的频率，此时周期是 24.025ns。同时由于开发板有一定的延迟，在设置输入延迟的时候设置 1ns 的延迟，见图 4.58。

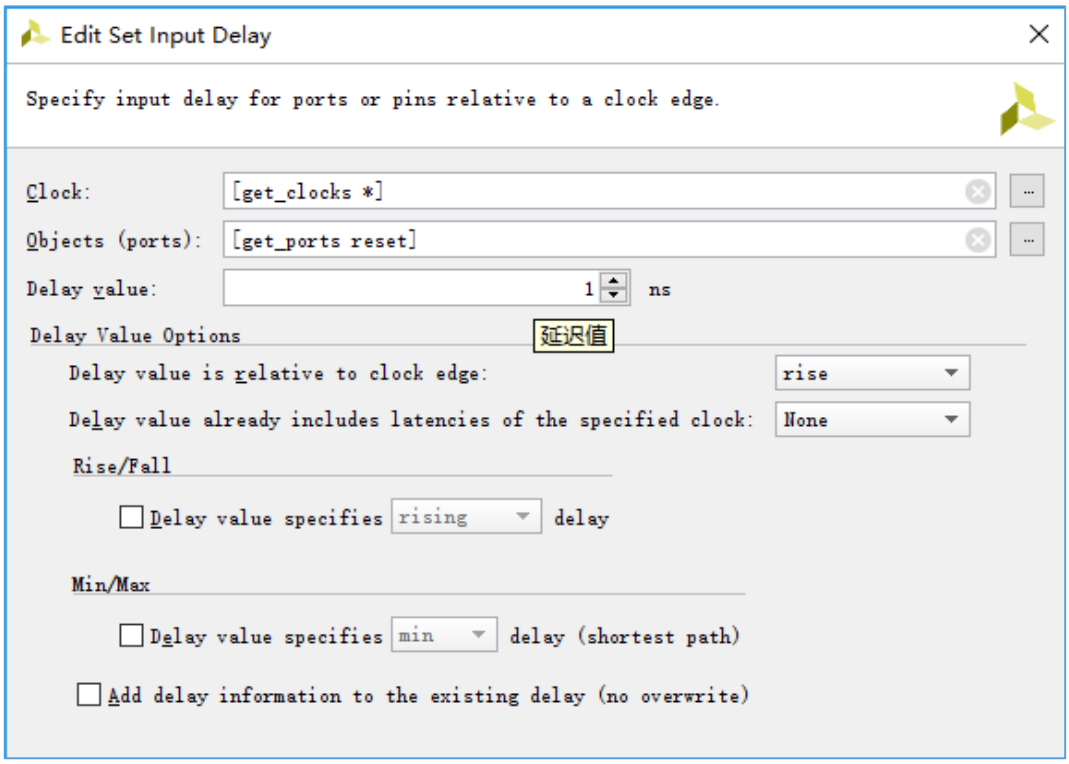

图 4.58 输入延迟设置

接下来用上文中提到的方法打开相应的界面，此时初次调试界面如图 4.59。

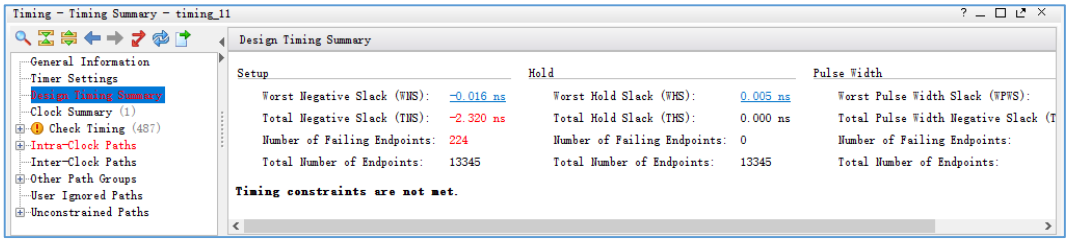

图 4.59 初次调试界面

由图 4.59 可以看到，设计上存在这一些违例，导致出现红色的警示数字，为此，我们可以点击图 4.60 左侧栏中的红色条目，展开至具体时钟线路。

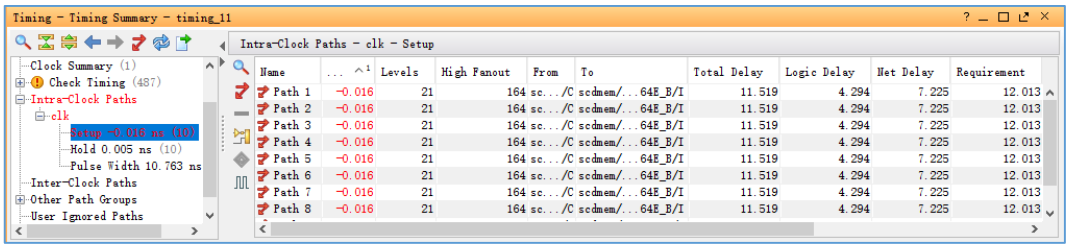

图 4.60 具体时钟线路

此时再双击右侧具体条目，可以看到如图 4.61 所示界面。

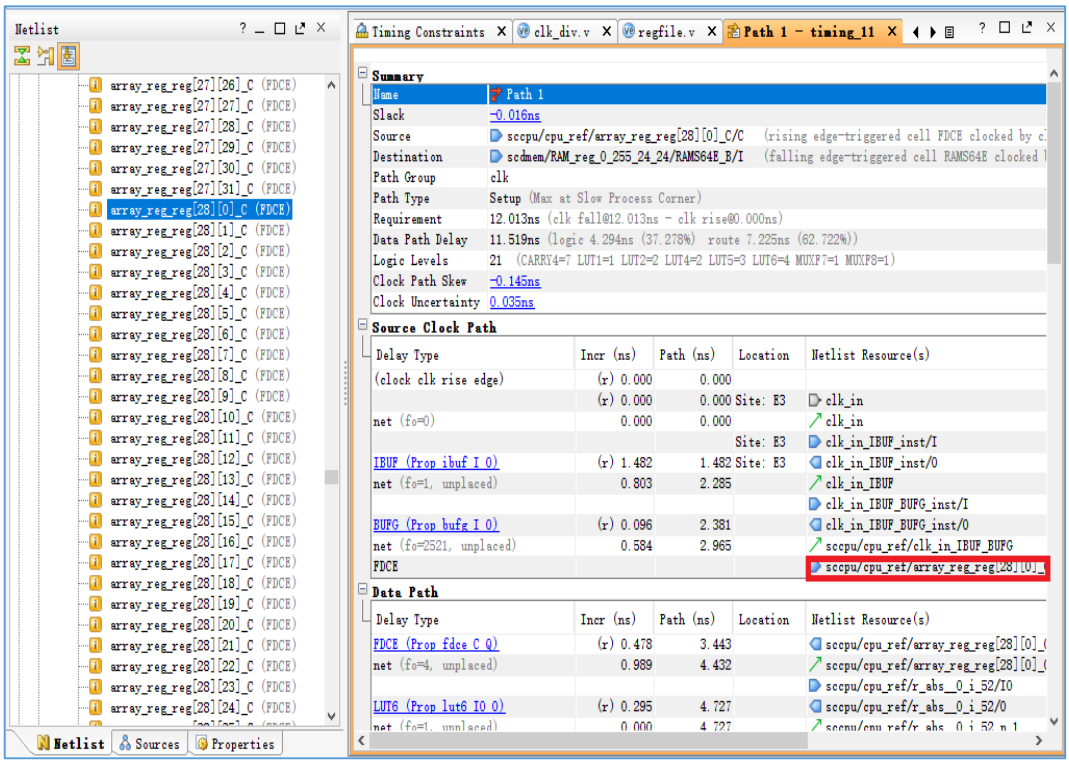

图 4.61 详细信息

从图 4.61 中左侧栏中高亮条目我们可以看出是 array_reg_reg[28][0]_C 这个变量出现了违例现象，并且可以从上图右侧中红色方框中看出变量的具体路径。此时我们找到相应代码，如图 4.62 所示。

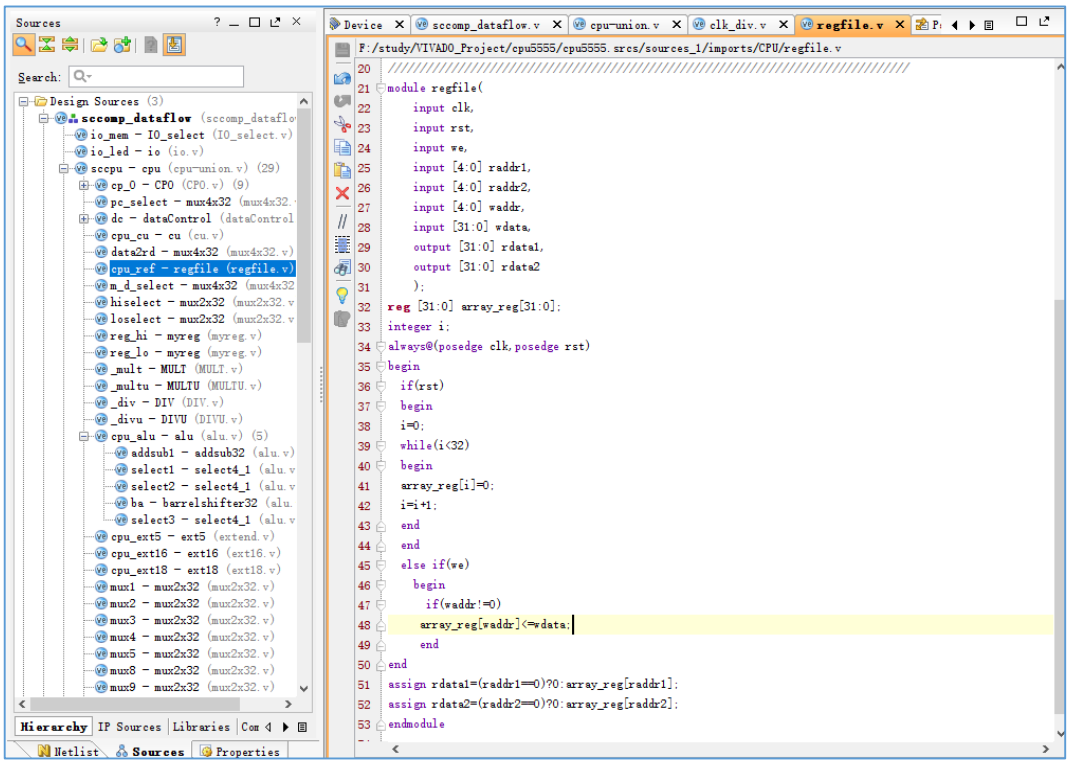

图 4.62 违例对应代码

此时我们发现 48 行的代码使用了非阻塞赋值，这会导致某时钟线路的 setup time

不足导致错误，故将其改为阻塞赋值。再重新综合工程进行时序分析，此时从图 4.63 中可以发现时序基本正常。

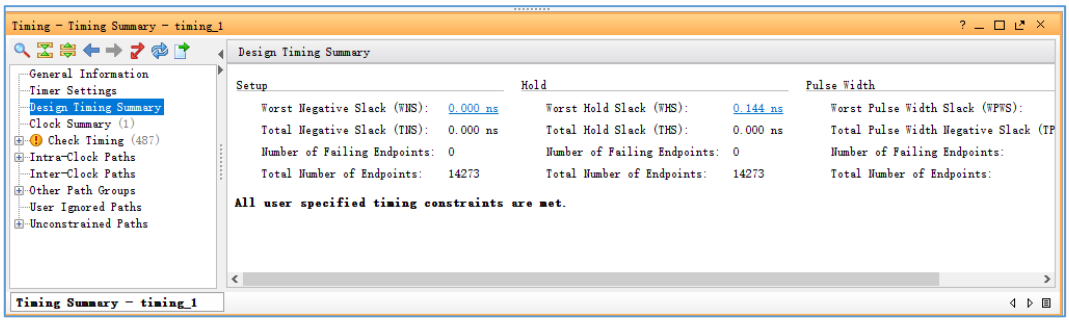

图 4.63 时序分析结果

4.5 Vivado 逻辑分析仪 ILA 的使用

逻辑分析仪是分析数字系统逻辑关系的仪器。逻辑分析仪是属于数据域测试仪器中的一种总线分析仪，同时对多条数据线上的数据流进行观察和测试的仪器，这种仪器对复杂的数字系统的测试和分析十分有效。逻辑分析仪用便于观察的形式显示出数字系统的运行情况，对数字系统进行分析和故障判断。

对于 FPGA 开发人员来说，逻辑分析仪是不可或缺的工具，当代码能够综合当代码能够综合、实现，但是烧写之后出现问题或者不能达到想要的效果，那么就需要 debug，逻辑分析仪就是 debug 过程中提高工作效率的利器。如果不使用逻辑分析仪抓取内部信号，那么我们就只能陷入"修改代码 查看现象 再修改代码 再查看现象"的循环，使用了逻辑分析仪就能快速定位问题。我们在进行计算机组成原理实验时，后面遇到的比较复杂的实验综合下板时间都较长，所以需要在下板时我们需要利用 vivado 逻辑分析仪来进行调试分析。本节继续通过流水灯实验，介绍如何利用 Vivado 内部功能强大的逻辑分析仪 ILA 来协助进行 FPGA 开发，实验流程如图 4.123 所示。

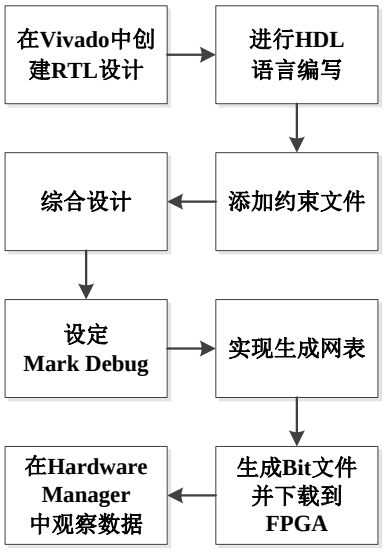

图 4.123 逻辑分析仪使用流程图

4.5.1 创建工程

1.如图 4.124 所示，创建一个新的工程“hw_debug”。

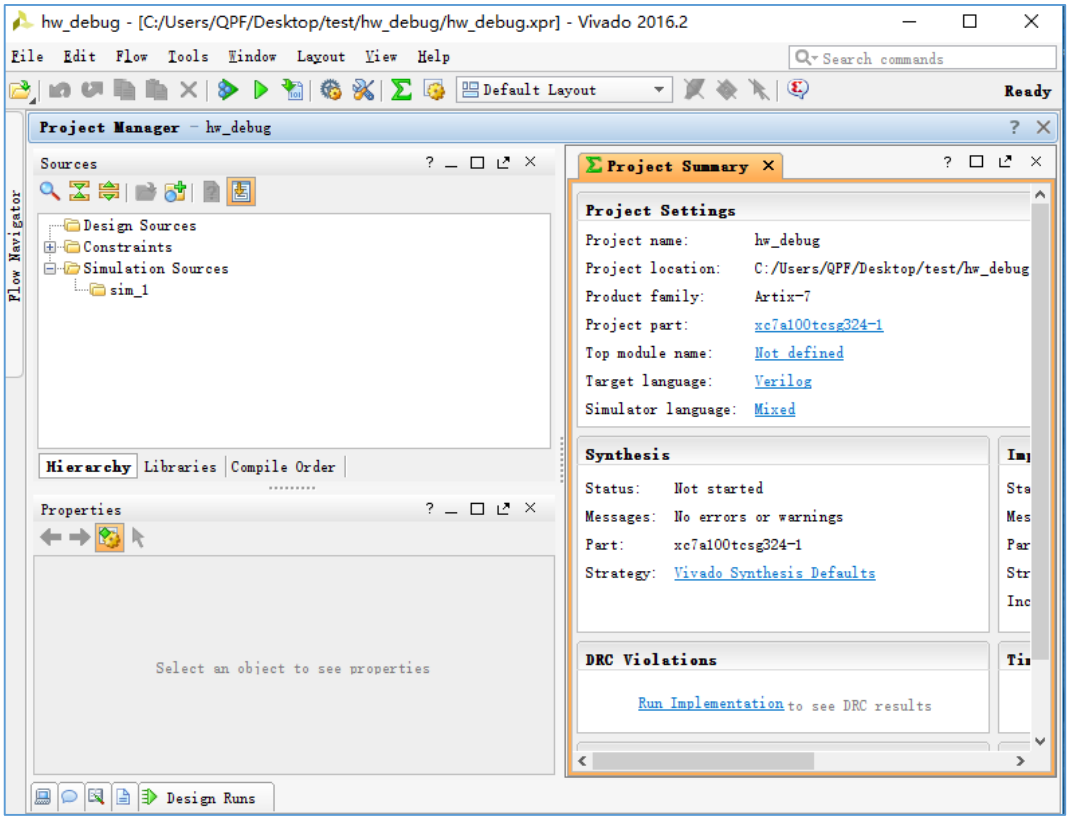

图 4.124 新建工程

4.5.2 添加源文件和约束文件

1.本次实验如图 4.125 所示创建三个.v 文件，分别是顶层模块“flow_led_top.v”和被调用的两个子模块“clk_div.v”和“flow_led.v”。

程序 4.5flow_led_top.v 代码

```
module  flow_led_top(
input clk,          //100MHz
output [15:0] led
    );
wire clk_pulse;
    clk_div clk_div(
        .clk(clk),
        .clk_pulse(clk_pulse)       //1Hz
        );

     flow_led flow_led(
        .clk(clk),
        .clk_pulse(clk_pulse),
```

```
        .led_r(led)
       );
endmodule
```

代码解释：顶层模块对两个子模块的调用

程序 4.6clk_div.v 代码

```
module  clk_div(
input clk,           //100MHz
output clk_pulse      //1Hz
    );
reg clk_pulse = 0;
reg [25:0]  div_counter = 0;

always @(posedge clk) begin
if(div_counter>=50000000) begin
         clk_pulse<=1;
         div_counter<=0;
end
else begin
         clk_pulse<=0;
         div_counter <= div_counter + 1;
end
end
endmodule
```

代码解释：输入为板上时钟，输出为 2hz 的脉冲信号。

程序 4.7flow_led.v 代码

```
module  flow_led(
input clk,
input clk_pulse,           //100MHz
   outputreg [15:0] led_r
    );

reg [15:0]  led_r = 16'h000f;

always @(posedge clk) begin
if(clk_pulse==1)
      led_r <= { led_r[11:0], led_r[15:12] };
else
      led_r <= led_r;
end
endmodule
```

代码解释：每收到一次脉冲信号，信号灯跳一次。

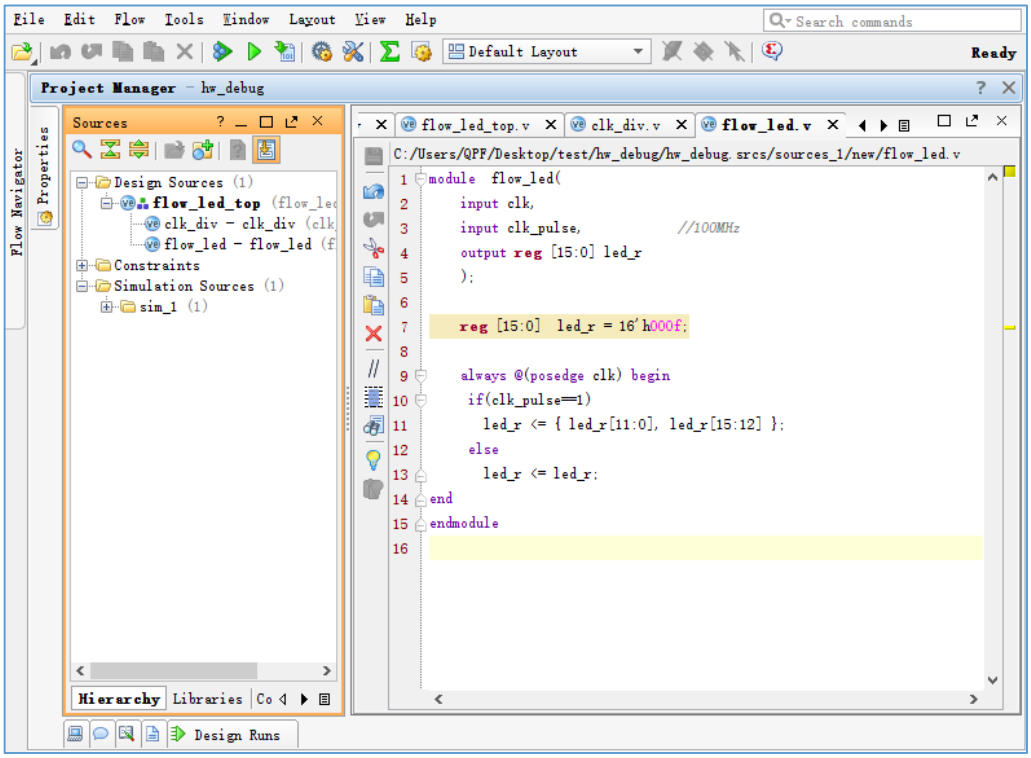

图 4.125 源文件

2.如图 4.126 所示，添加如程序 4.8 所示约束文件“flow_led.xdc”。

程序 4.8 flow_led.xdc 约束文件

```
set_property IOSTANDARD LVCMOS33 [get_ports {led[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property PACKAGE_PIN H17 [get_ports {led[15]}]
set_property PACKAGE_PIN K15 [get_ports {led[14]}]
set_property PACKAGE_PIN N14 [get_ports {led[13]}]
set_property PACKAGE_PIN J13 [get_ports {led[12]}]
```

```
set_property PACKAGE_PIN R18 [get_ports {led[11]}]
set_property PACKAGE_PIN V17 [get_ports {led[10]}]
set_property PACKAGE_PIN U17 [get_ports {led[9]}]
set_property PACKAGE_PIN V16 [get_ports {led[8]}]
set_property PACKAGE_PIN U16 [get_ports {led[7]}]
set_property PACKAGE_PIN T15 [get_ports {led[6]}]
set_property PACKAGE_PIN U14 [get_ports {led[5]}]
set_property PACKAGE_PIN T16 [get_ports {led[4]}]
set_property PACKAGE_PIN V15 [get_ports {led[3]}]
set_property PACKAGE_PIN V14 [get_ports {led[2]}]
set_property PACKAGE_PIN V12 [get_ports {led[1]}]
set_property PACKAGE_PIN V11 [get_ports {led[0]}]
set_property PACKAGE_PIN E3 [get_ports clk]
```

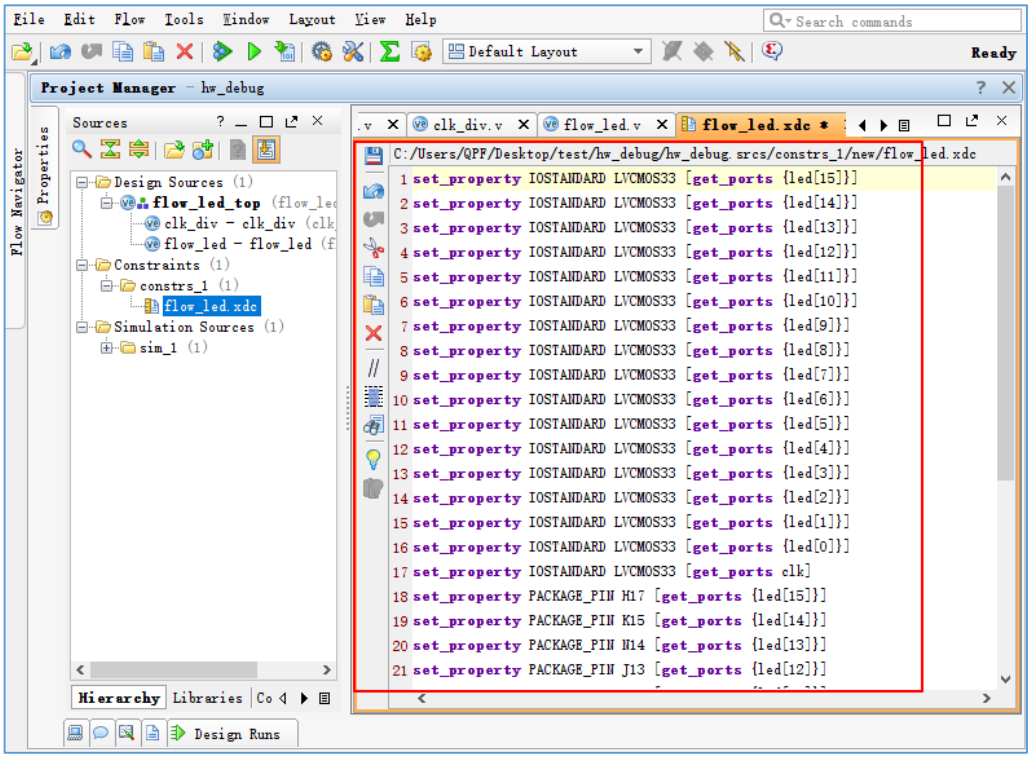

图 4.126 约束文件

4.5.3 综合

1 .如图 4.127 所示, 在“Flow Navigator”栏中的“Synthesis”下点击“Run Synthesis”。右上角的进度条“Running synth_design”指示正在对工程进行综合。综合完成之后在弹出的对话框中点击“Cancel”取消。在“Flow Navigator”一栏中，找到“Synthesis”->“Open Synthesised Design”->“Schematic”,点击“Schematic”，结果见图 4.128。

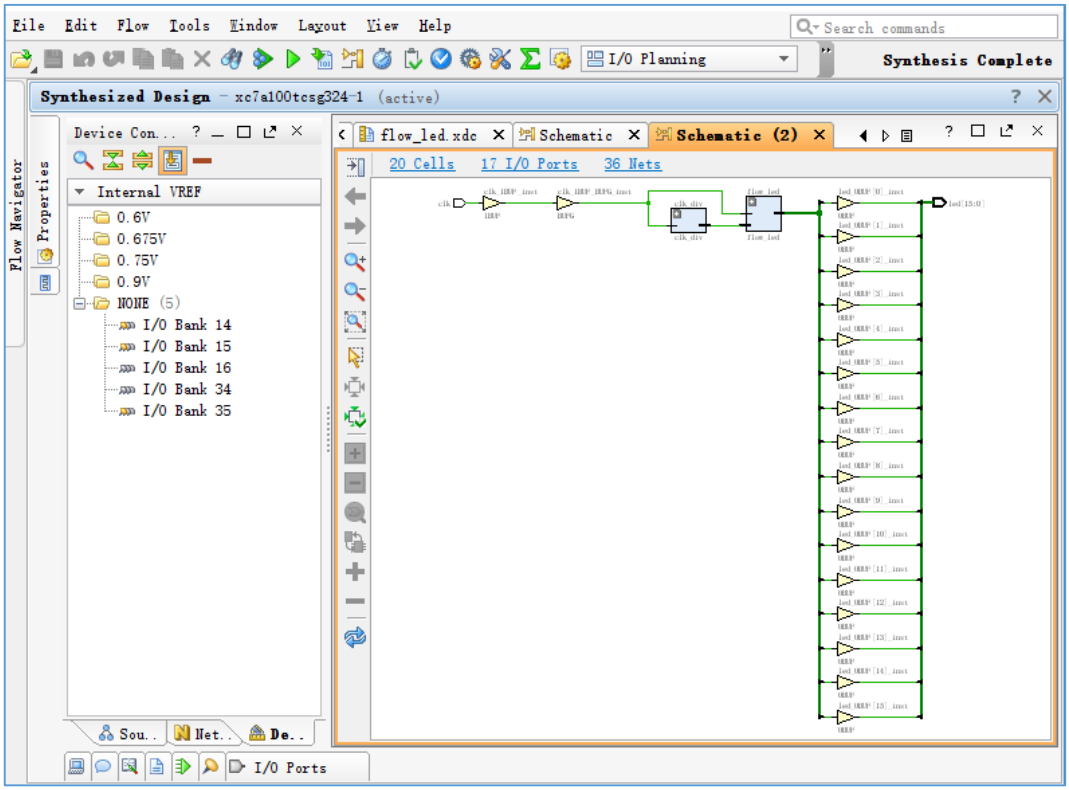

图 4.127Synthesis 选项

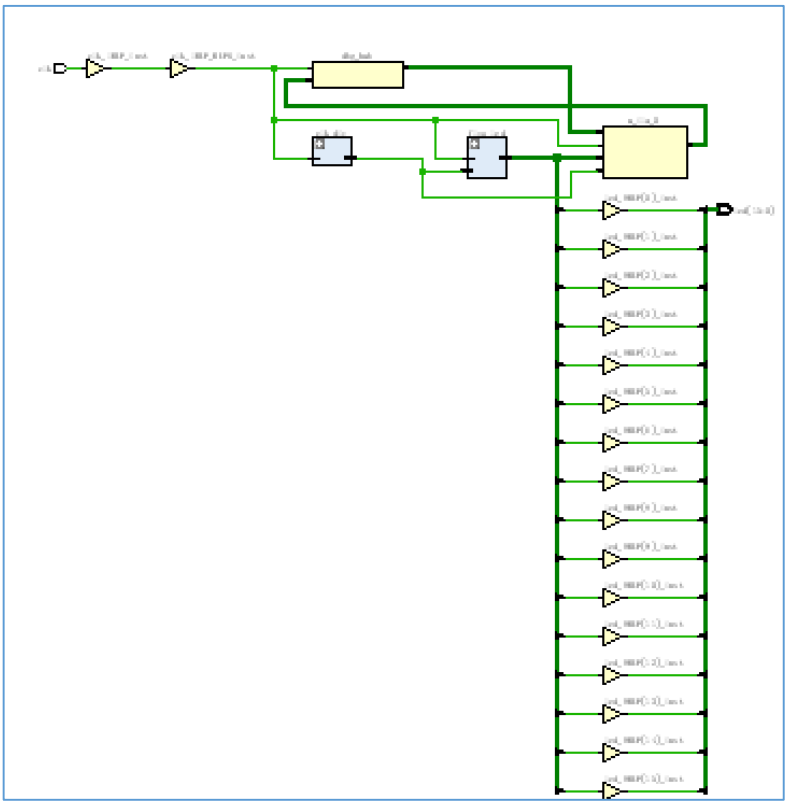

图 4.128 综合结果

4.5.4 Mark Debug

1.我们先 Mark 流水灯模块的输入线。如图 4.129 所示，在“Schematic”标签页中，点击左侧工具栏中的放大镜图标，将电路图放大到合适大小。找到“clk_div”模块和“flow_led”模块之间的连线“clk_pulse”，选中后右击，选择“Mark Debug”。

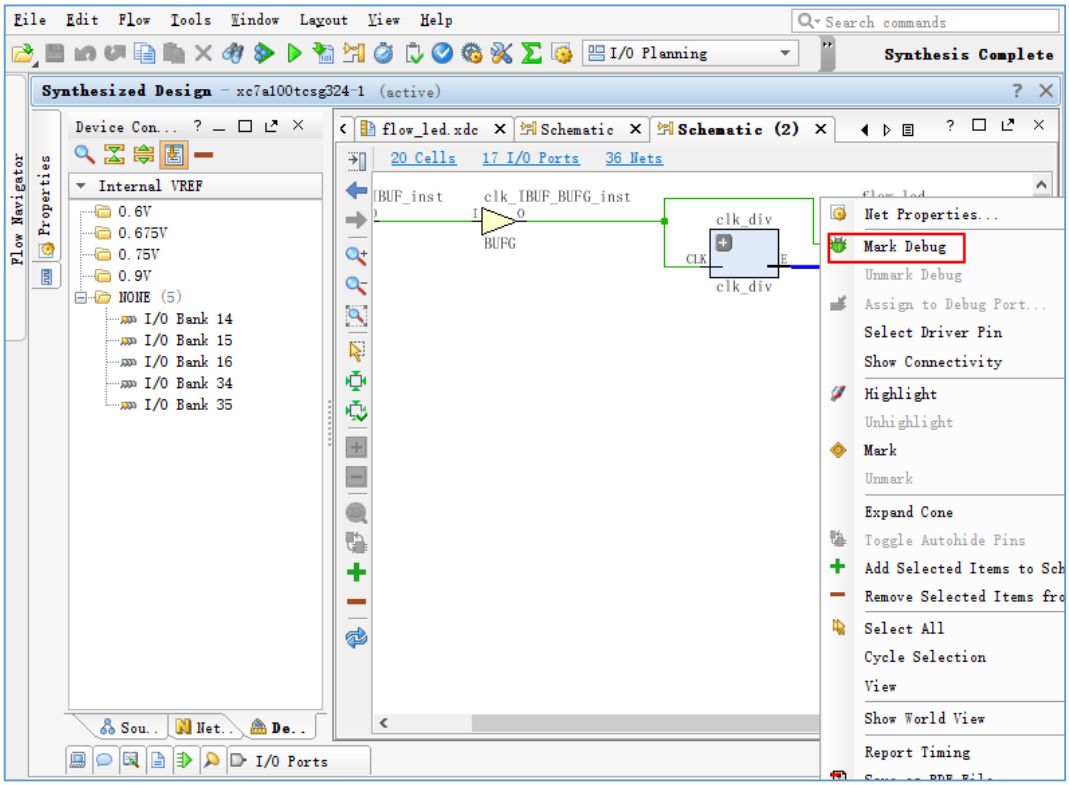

图 4.129Mark Debug 标签

2.然后我们再 Mark 流水灯模块的输出线。图 4.130 所示，找到与“flow_led”模块的输出端口相连的信号线“led_OBUF”，选中后右击，选择“Mark Debug”。

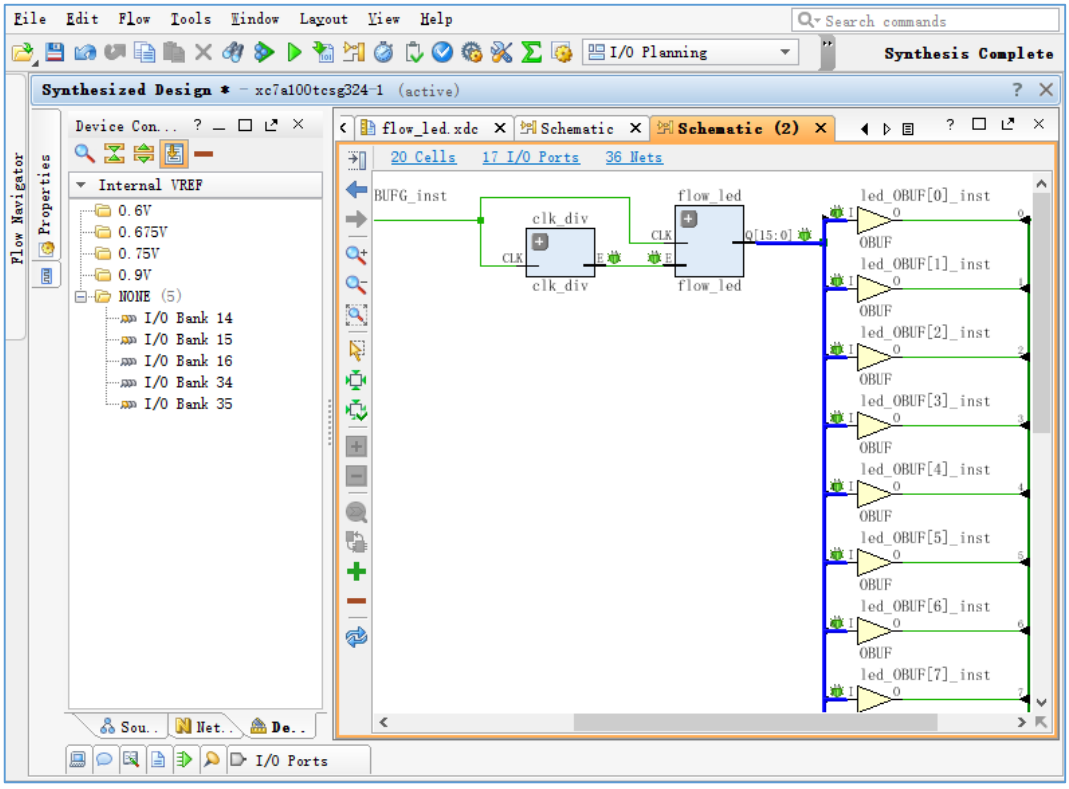

图 4.130 流水灯模块

在这个实验中我们之所以 Mark 这两个信号线，是因为我们的我们要关注核心部件流水灯模块在指定的输入信号下会不会有理想的结果。

4.5.5 Set up Debug

1.如图 4.131 所示，在“Debug”窗口中(可通过在菜单栏中点击“Layout”->“Debug”打开)，单击选中“Unassigned Debug Nets”，然后右击选择“Set Up Debug”。

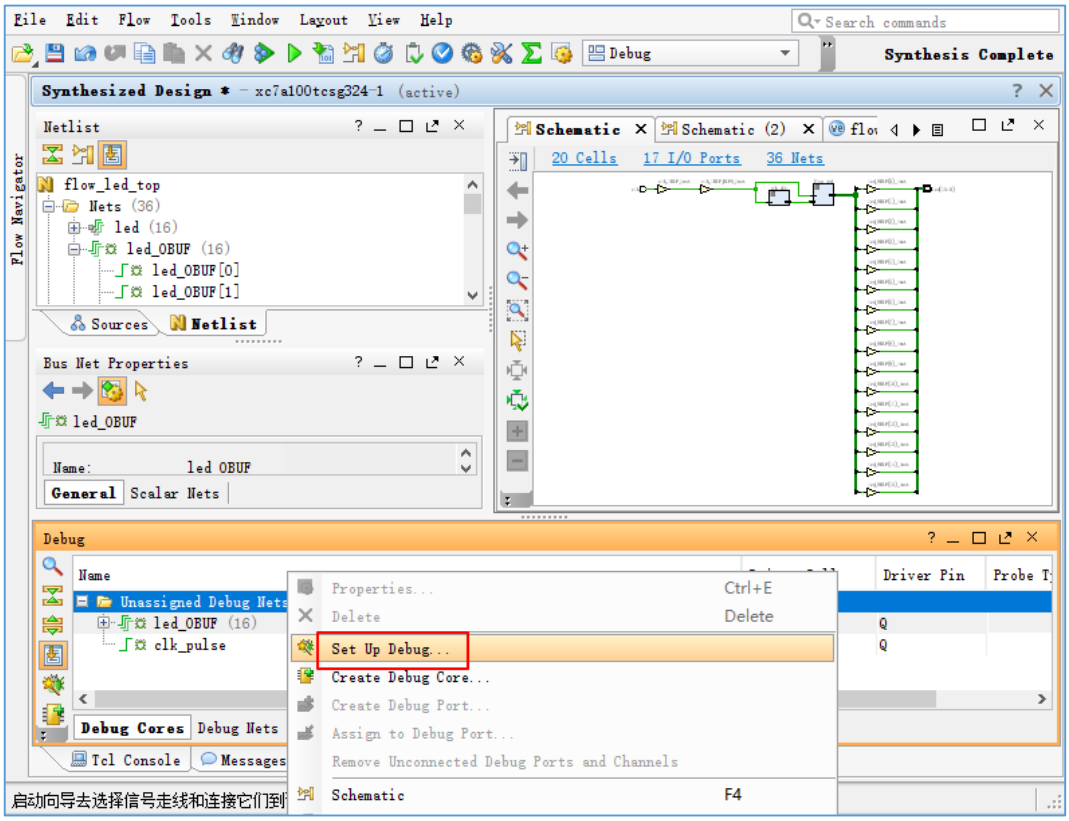

图 4.131Set Up Debug 选项

2.如图 4.132 所示，在“Set Up Debug”向导中连续点击“Next”，最后点击“Finish”。

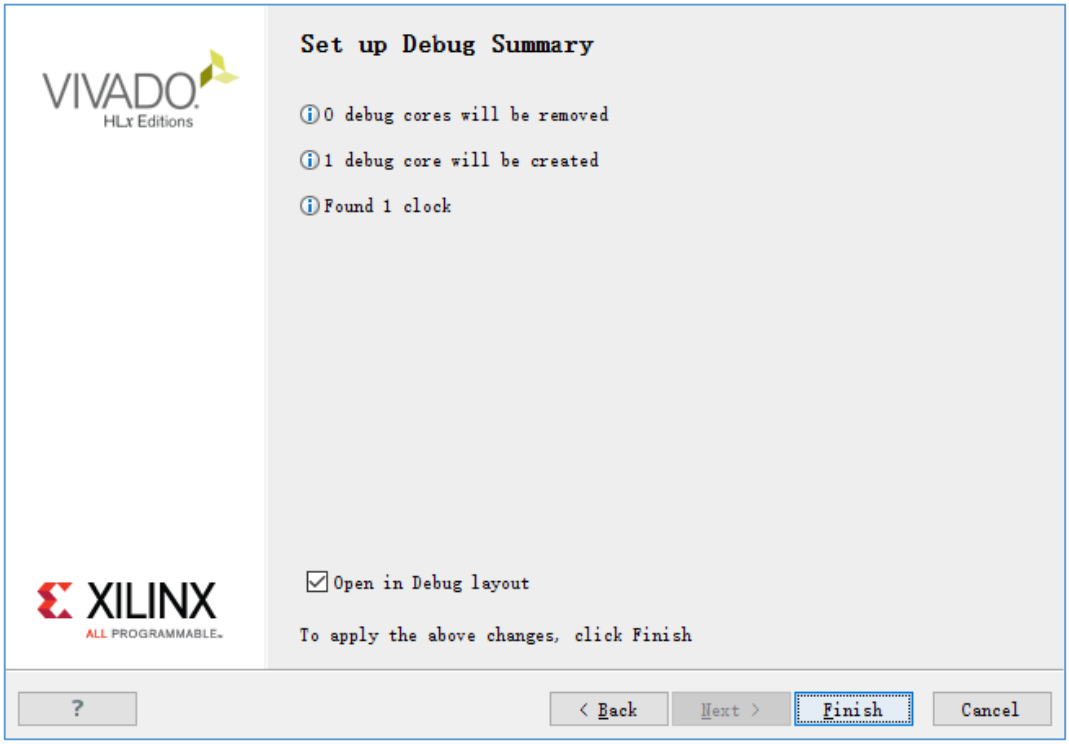

图 4.132Set Up Debug 向导

3.如图 4.133 所示，在菜单栏中点击“File”->“Save Constraints”。在弹出的对话框中点击“OK”。然后在“Source”窗口中打开“flow_led.xdc”，在文档的底部可以看到“Mark Debug”及“Set Up Debug”的相关信息被添加上去了。

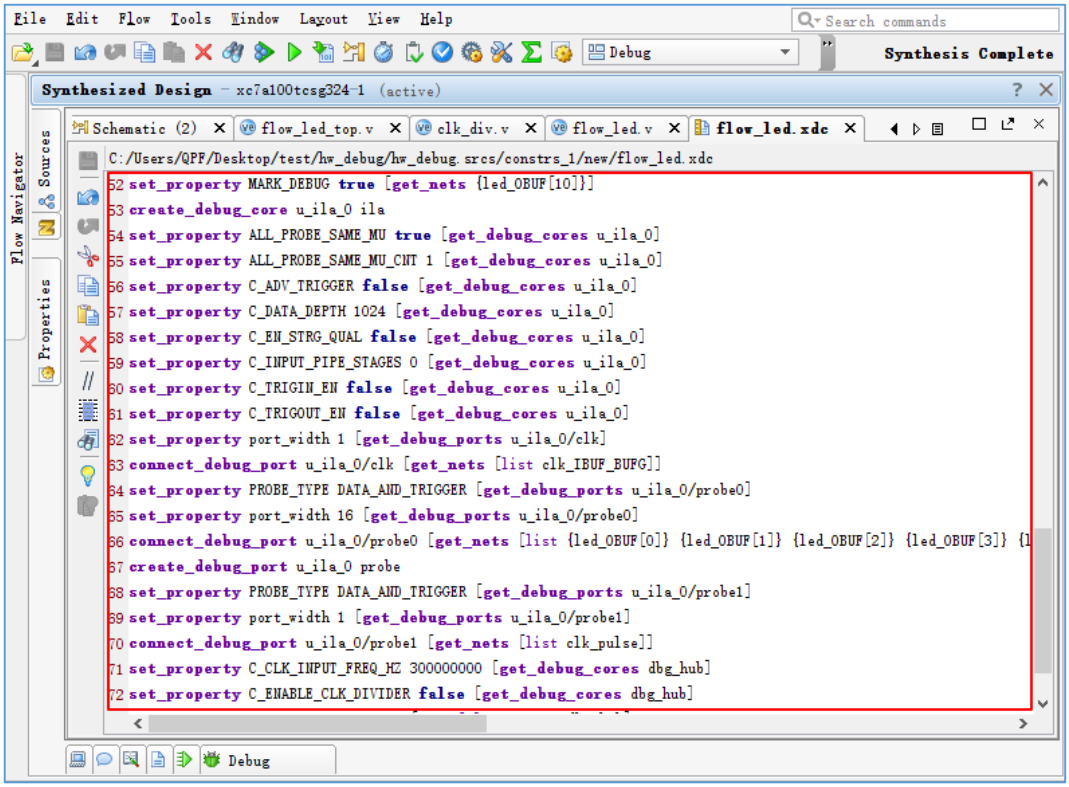

图 4.133XDC 文件

4.5.6 生成 Bit 文件

1.在“Flow Navigator”一栏中的“Program and Debug”下点击“Generate Bitstream”,此时会提示工程没有实现，点击“Yes”,会自动执行实现过程。

4.5.7 下载

1.用 Micro USB 线连接电脑与板卡上的 JTAG 端口，打开电源开关。
2.生成比特流文件完成后，打开“Hardware Manager”。在“Hardware Manager”界面点击“Open target”,选择“Auto Connect”。连接成功后，在目标芯片上右击，选择“Program Device”。在弹出的对话框中“Bitstream File”一栏已经自动加载本工程生成的比特流文件，点击“Program”对 FPGA 芯片进行编程，见图 4.134。

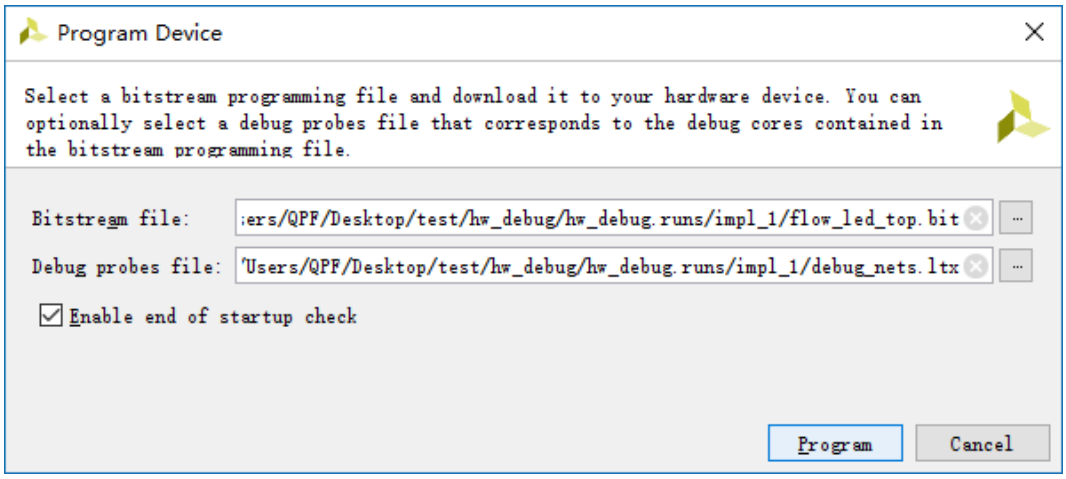

图 4.134FPGA 芯片编程

3.如图 4.135 所示，下载完成后在板卡上可观察到四位 led 灯为一组从右向左循环点亮。

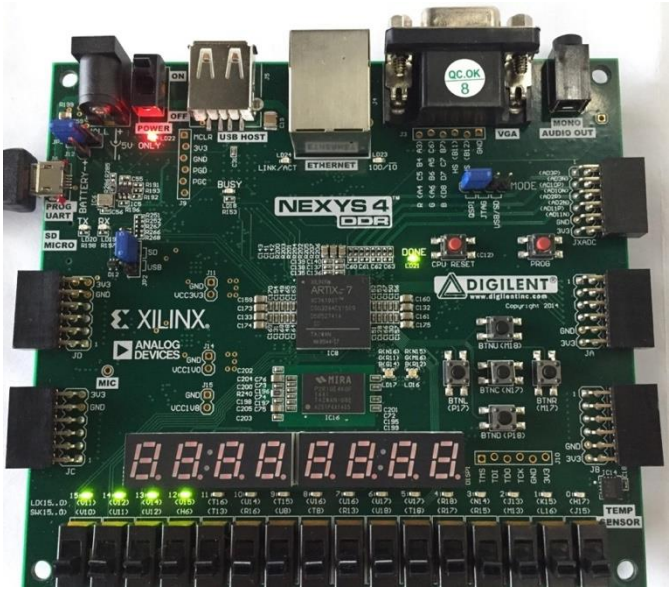

图 4.5.13 下载完成后的板卡

4.下载完成后“Hardware Manager”界面如图 4.136 所示。

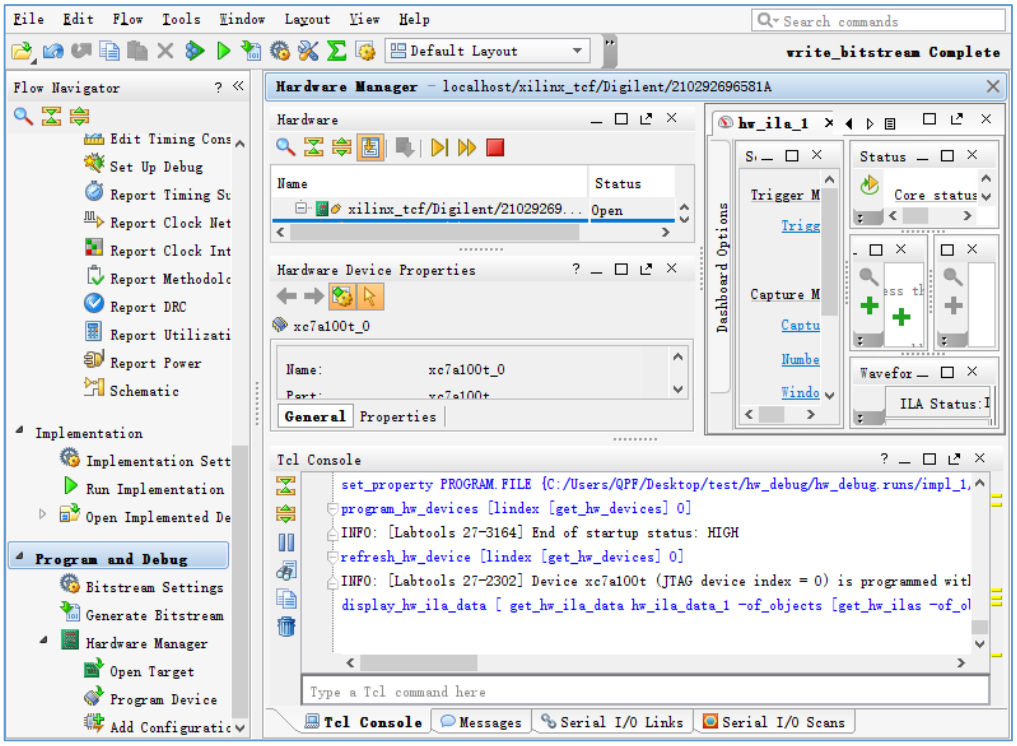

图 4.136Hardware Manager 界面

4.5.8 Hardware Debug

1.如图 4.137 所示，选中目标芯片，点击上方的“Run Trigger Immediate”,在波形窗口中可看到触发信号。

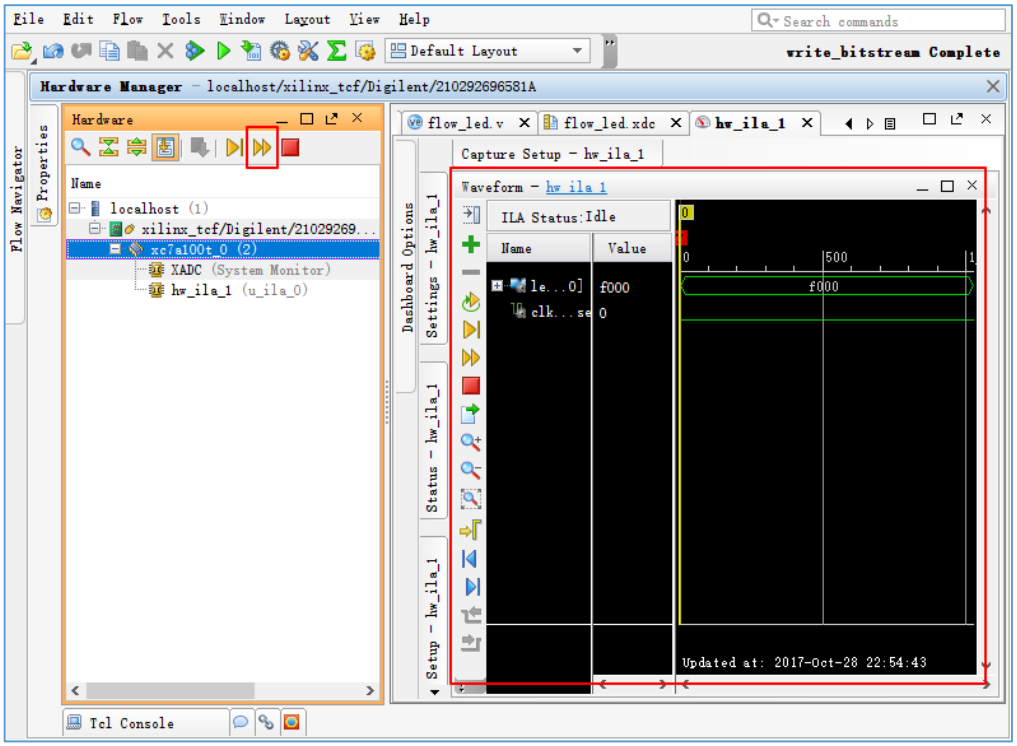

图 4.137Run Trigger Immediate 选项

2.在图 4.138 所示的“Trigger Setup”窗口中点击加号，点击加号后出现的信号双击即

可加入窗口，这些为将要进行调试的信号。选中图 4.5.17 中的“clk_pulse”，双击添加到窗口中。

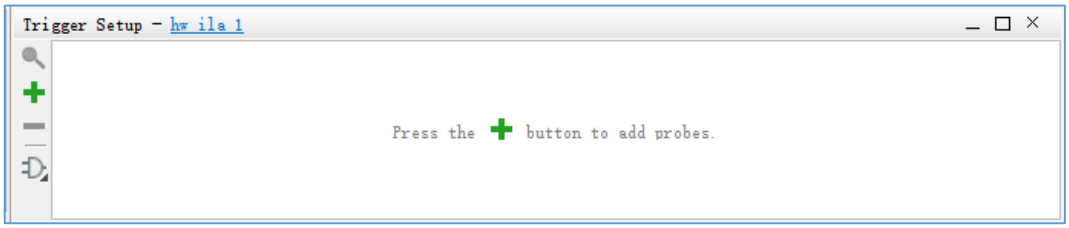

图 4.138Trigger Setup 窗口

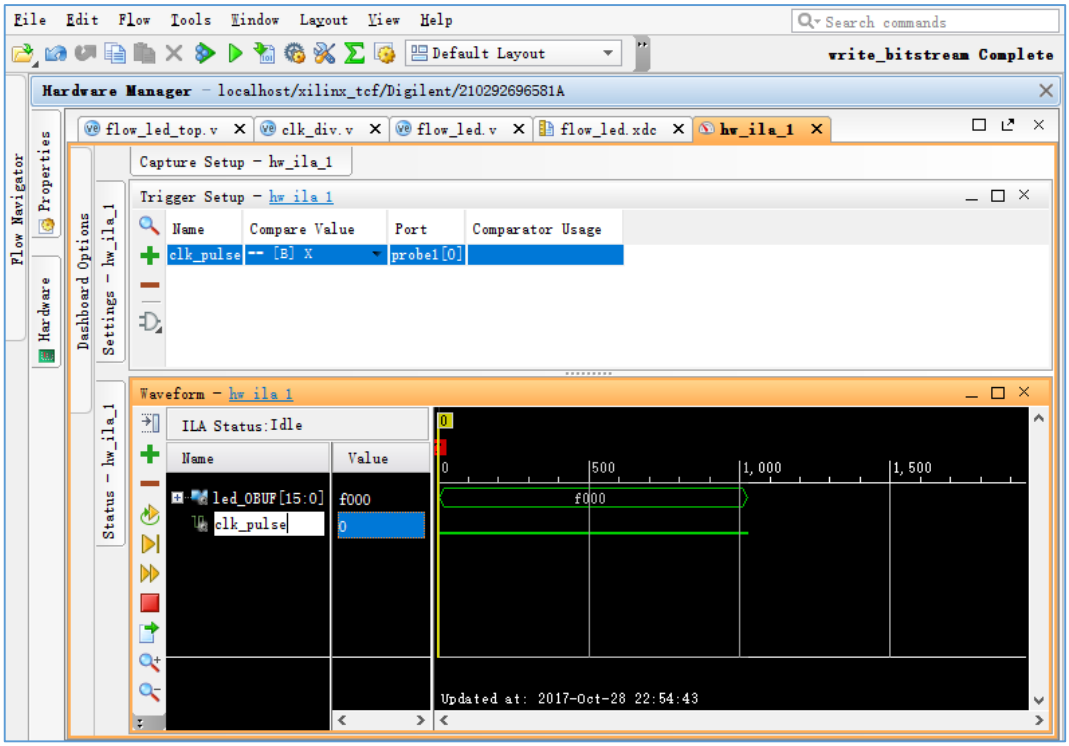

图 4.139clk_pulse 选项

3.设置要调试的信号的值。我们需要查看在 clk_pulse 信号给了一个输入时，led_OBUF 的响应情况。那么我们在调试时可以设定值。如图 4.140 所示，在“Trigger Setup”窗口中将“led_OBUF”的“Compare Value”设为“[H] 00F0”，将“clk_pulse”的“Compare Value”设为 1。意思就是先给 led_OBUF 一个初始值 00F0，给定 clk_pulse 的触发值为 1。那么我们就要看当 clk_pulse 触发时的波形变化。

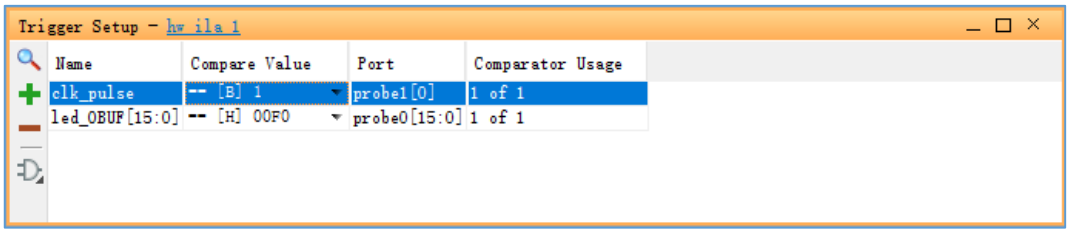

图 4.140 调试信号值设置

4.在图 4.141 所示的波形窗口中添加要观察信号的波形。

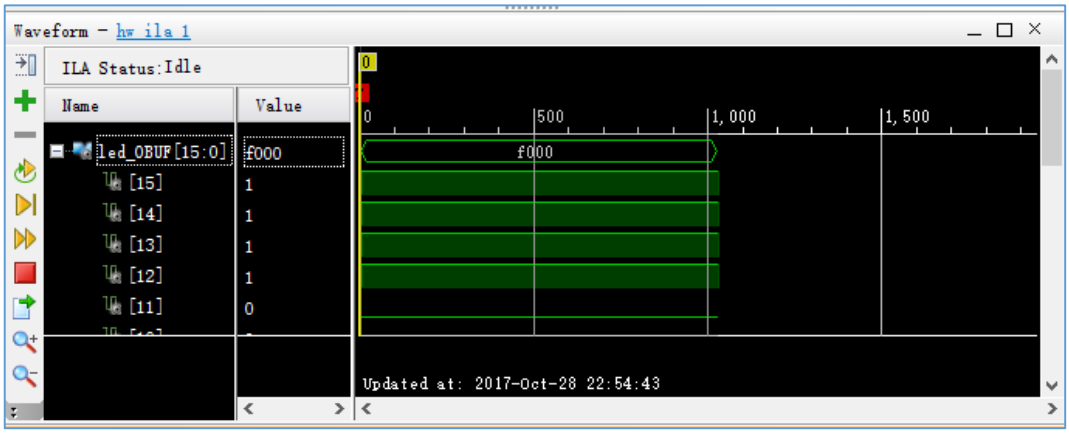

图 4.141 波形窗口

如图 4.142 所示，通过在“Waveform”窗口点击加号，在“Add Probes”列表中双击要添加的信号添加到波形窗口中。

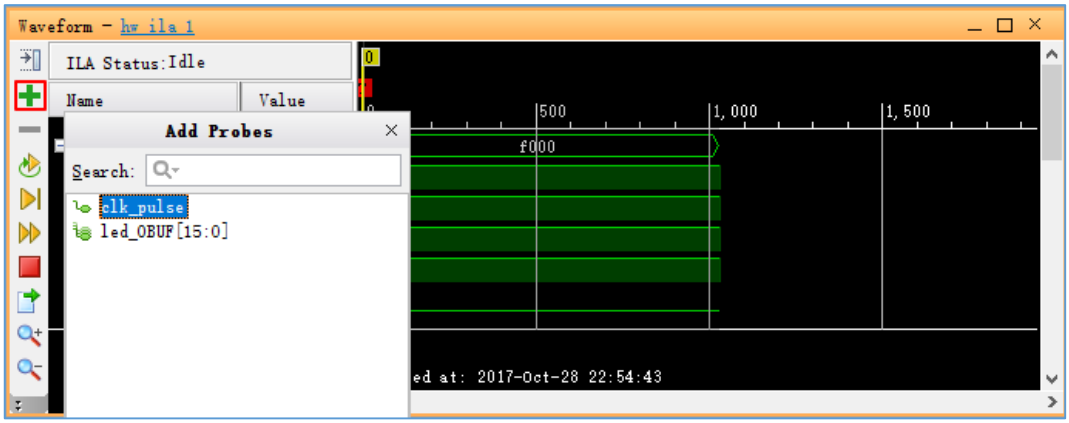

图 4.142 添加信号波形

5．在图 4.143 所示的“hw_ila_1”的“Settings”窗口中，“Trigger position in window”一栏可以设置触发的位置，那么我们填个 200 看看结果触发时间的设置。

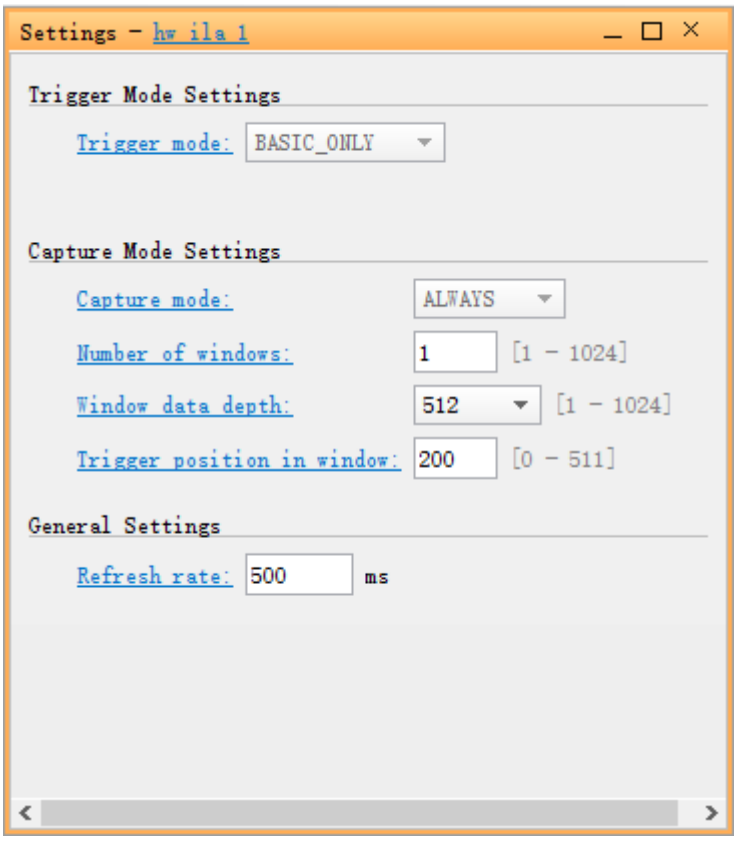

图 4.143Settings 窗口

6.观察结果。在“Hardware”窗口选中“hw_ila_1”,然后点击上方的“Run Trigger”按钮。如图 4.144 所示，在“Status”窗口可观察到状态由“Idle”跳转到“Waiting for Trigger”。当状态跳转到“Full”后回到“Idle”状态，此时将波形图中红色竖线标注出了触发的时刻。

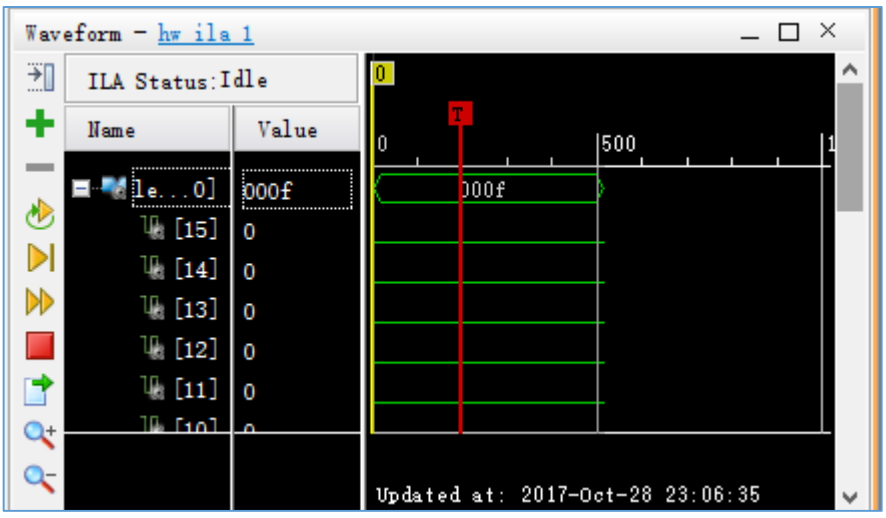

图 4.144Status 窗口

右击波形，点击放大选项，我们可以发现在 200 时，pulse 跳 1 时，led_OBUF 也跟预期的一样由 00f0 变为 0f00。如果结果与预期不符的话，我们要去检查自己写的模块，看检查相应时序是否正确。下面以一个写信号为高电平有效的 32*8 的存储器为例说明，代码如程序 4.9 所示。

程序 4.9 存储器模块举例

```
module dmem (
input clk,
input DM_W,
input [4:0] addr,
input [7:0] wdata,
output[7:0] rdata
    );
reg [7:0] RAM[31:0];
assign rdata = RAM[addr] ;
always@(negedge clk  )
begin
if(DM_W)
    begin
        RAM[addr] <= wdata;
    end
end
endmodule
```

我们写一个写信号触发模块(功能:在给定的输入信号变换一次时输出一个脉冲信号):

程序 4.10 写信号触发模块

```
module DM_W_div(
input clk,
input DM_W_switch,
output DM_W_pulse
    );
reg DM_W_switch_delay;

always@(posedge clk )
begin
       DM_W_switch_delay <= DM_W_switch;
end
assign DM_W_pulse = DM_W_switch ^ DM_W_switch_delay;

endmodule
```

我们根据上面的逻辑分析仪使用教学，将与存储器模块相关的 DM_W,addr,wdata,rdata 都作为 debug 的观测对象，约束文件中我将地址输入、数据输入约束到开关上，将读数据约束到 led 灯上。如图 4.145 所示，将 triggersetup 中触发条件设置为写信号为 1 时然后拨动触发开关（此时我将写信号触发信号约束到板上一个开关上，拨一次触发一次）观察波形。

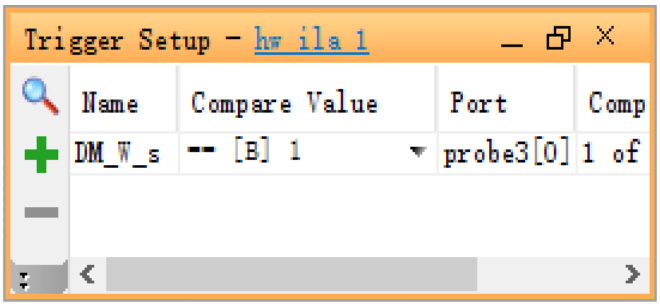

图 4.145led 灯数据约束

我们得到如图 4.146 所示波形：

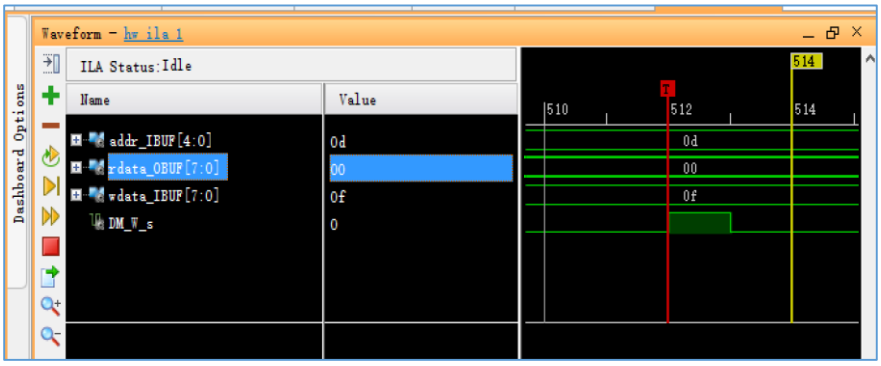

图 4.146 波形图

这说明存储器并没有单元写入了数据，时序有错误。我们检查自己写的模块，将下降沿改为上升沿试试。那么接下来我们将模块改为上升沿触发，代码如程序 4.11 所示：

程序 4.11 上升沿触发

```
module dmem (
input clk,
input DM_W,
input [4:0] addr,
input [7:0] wdata,
output[7:0] rdata
    );
reg [7:0] RAM[31:0];
assign rdata = RAM[addr] ;
always@(posedge clk  )
begin
if(DM_W)
    begin
        RAM[addr] <= wdata;
    end
end
endmodule
```

同样按上面步骤操作，得到如图 4.147 波形：

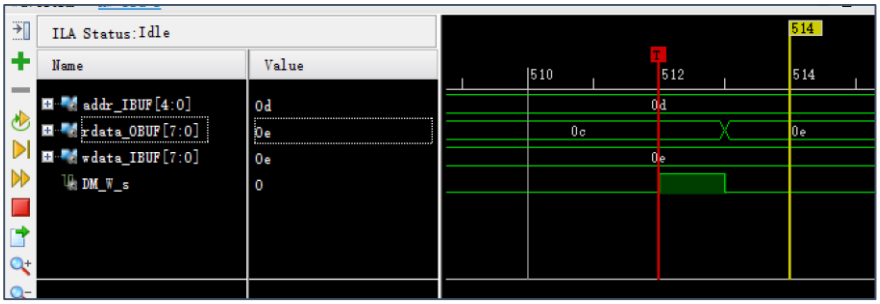

图 4.147 波形图

我们发现存储器按照预期地在写信号来时写入了数据，说明改动正确。以后我们检查自己写的模块时可以按照以上例子的方法操作。