

静态时序分析

1. 概念

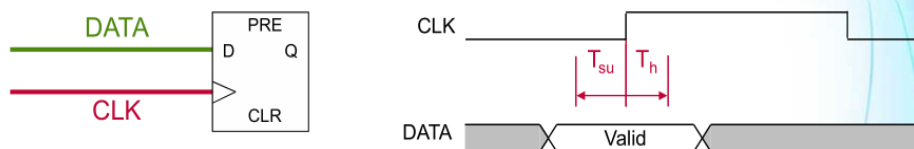
静态时序分析（Static Timing Analysis, STA），是对数字电路的时序进行计算、预计的工作流程，该流程不需要通过输入激励的方式进行仿真。

静态时序分析的前提是设计者提出要求，然后利用时序分析工具、根据特定的时序模型进行分析，给出时序报告。进行静态时序分析，主要目的是为了提高系统工作主频以及增加系统的稳定性。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。

在综合后或者布局布线后两个设计阶段都可以查看时序报告，区别在于综合后时序分析的走线延迟（Net Delay）基于经验预估值，而布局布线后的时序分析则基于真实的走线，更加精确。

2. 参数和术语

Setup & Hold

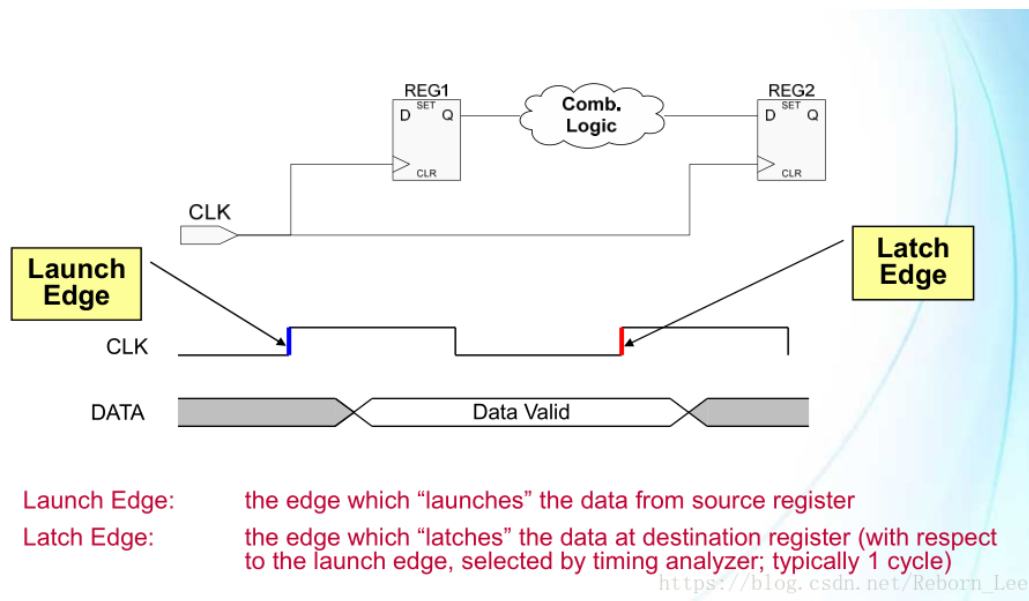


Setup: The minimum time data signal must be stable BEFORE clock edge

Hold: The minimum time data signal must be stable AFTER clock edge

Together, the setup time and hold time form a Data Required Window, the time around a clock edge in which data must be stable.

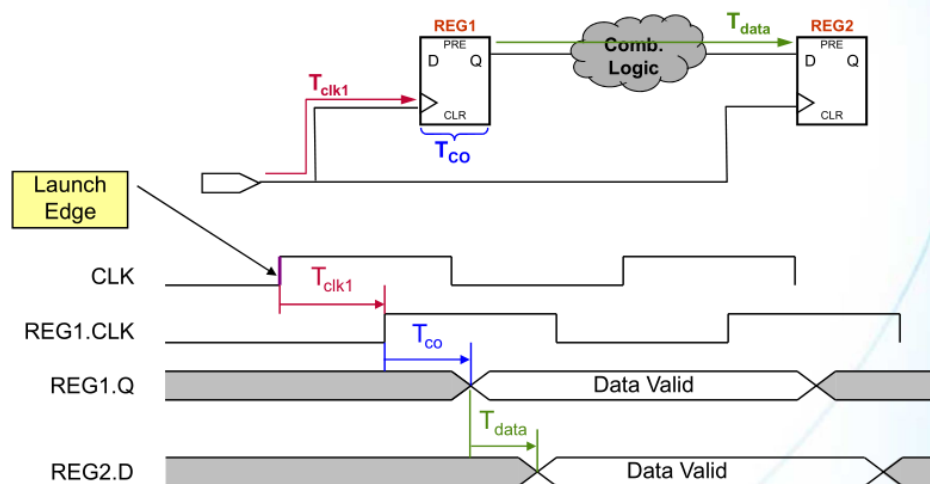
- 建立时间（Setup Time）：时钟有效沿到来之前，数据必须保持稳定的最小时间。
- 保持时间（Hold Time）：时钟有效沿到来之后，数据必须保持稳定的最小时间。



- 发射沿（Launch Edge）：第一级寄存器数据变化的时钟边沿，也是静态时序分析的起点。
- 锁存沿（Latch Edge）：数据锁存的时钟边沿，也是静态时序分析的终点。

Data Arrival Time

- The time for data to arrive at destination register's D input



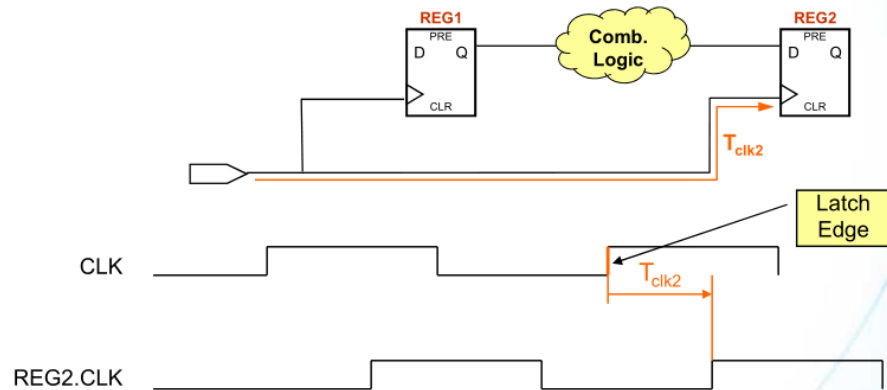
$$\text{Data Arrival Time} = \text{launch edge} + T_{clk1} + T_{co} + T_{data}$$

https://blog.csdn.net/Reborn_Lee

- 数据到达时间（Data Arrival Time）：时钟到达寄存器的时钟输入端的延迟+数据从时钟有效沿开始到输出之间的延迟+数据经过组合逻辑以及布线延迟到达目的寄存器输入端的延迟。

Clock Arrival Time

- The time for clock to arrive at destination register's clock input



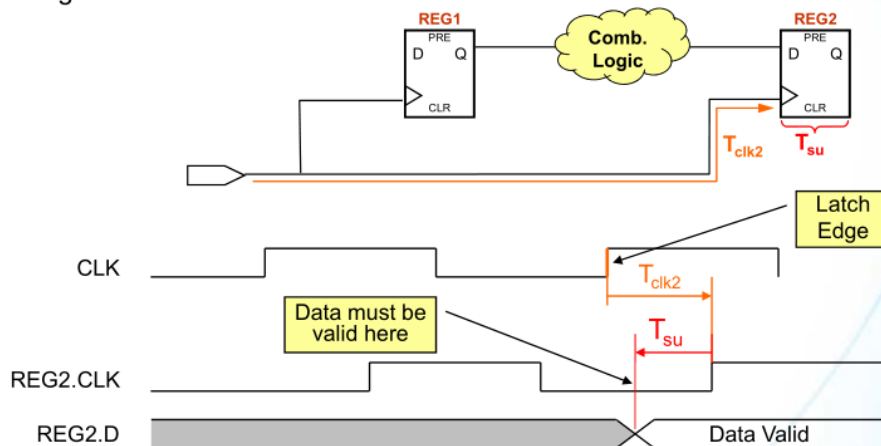
$$\text{Clock Arrival Time} = \text{latch edge} + T_{clk2}$$

https://blog.csdn.net/Reborn_Lee

- 时钟到达时间（Clock Arrival Time）：时钟到达捕获寄存器与时钟输入端之间的延迟。

Data Required Time - Setup

- The minimum time required for the data to get latched into the destination register



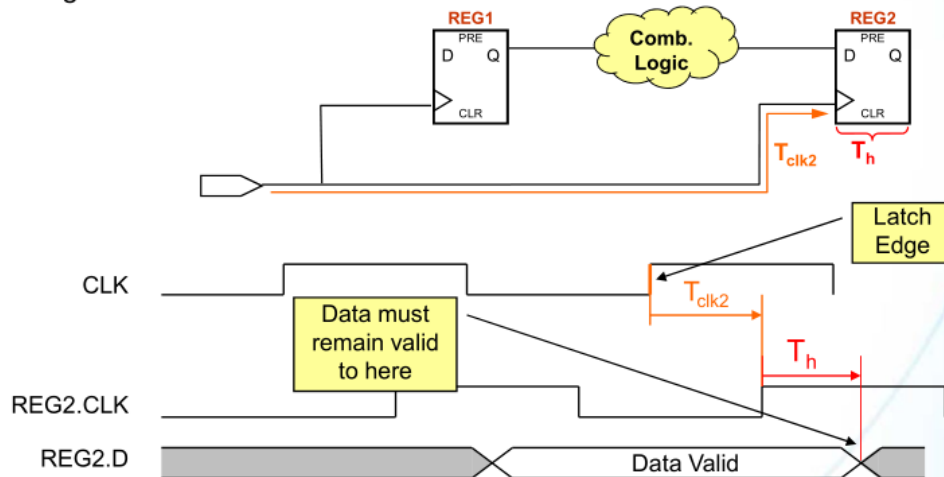
$$\text{Data Required Time} = \text{Clock Arrival Time} - T_{su} - \text{Setup Uncertainty}$$

http://blog.csdn.net/Reborn_Lee

- 数据需求时间（建立）（Data Required Time - Set up）：数据在目的寄存器锁存所需要的时间。

Data Required Time - Hold

- The minimum time required for the data to get latched into the destination register

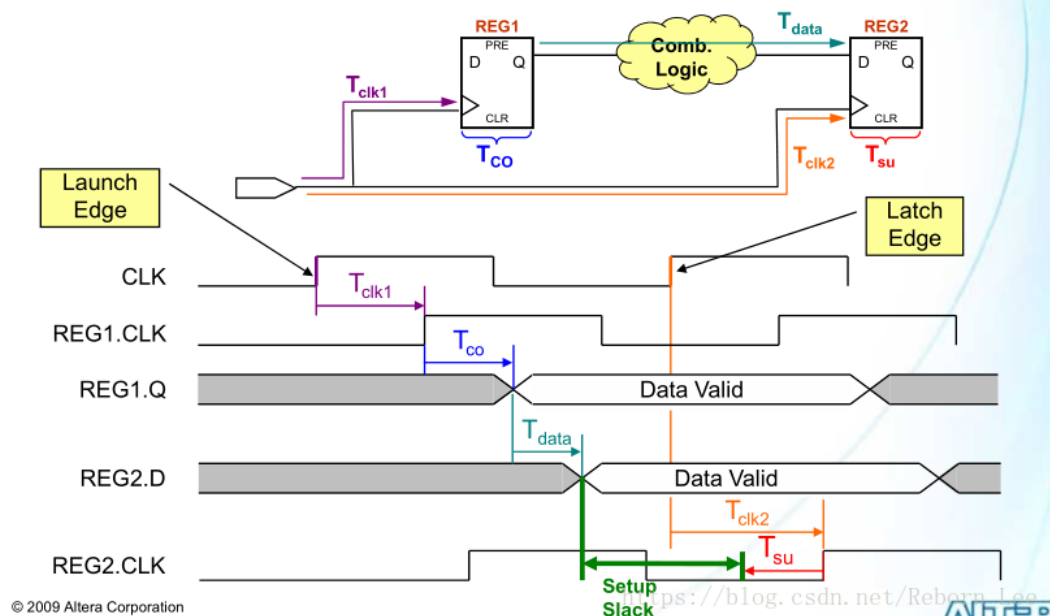


$$\text{Data Required Time} = \text{Clock Arrival Time} + T_h + \text{Hold Uncertainty}$$

- 数据需求时间（保持）（Data Required Time - Hold）：数据在目的寄存器保持稳定所需要的最小时间。

Setup Slack

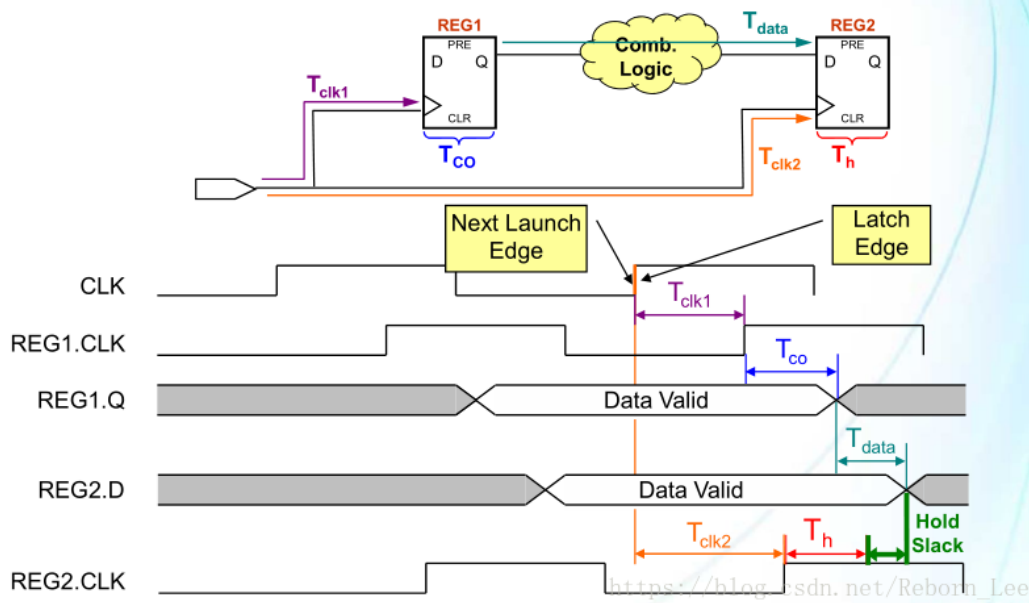
- The margin by which the setup timing requirement is met. It ensures launched data arrives in time to meet the latching requirement.



- 建立时间裕量（Setup Slack）：等于“数据需求时间（建立）”减去数据到达时间，若大于零，则满足时序要求。

Hold Slack

- The margin by which the hold timing requirement is met. It ensures latch data is not corrupted by data from another launch edge.



- 保持时间裕量（Hold Slack）：等于“数据需求时间（保持）”减去数据到达时间，若大于零，则满足时序要求。

3. XDC 约束

Vivado 不仅支持在上述图形化界面中添加时钟约束，还支持手动撰写 XDC 约束文件，包括添加外部时钟约束、建立时钟组、管脚分配等等。XDC 是 Xilinx Design Constraints 的简写，但其基础语法来源于业界统一的约束规范 SDC（最早由 Synopsys 公司提出，故名 Synopsys Design Constraints）。

XDC 可以作为一个整体文件被工具读入，也可以在实现过程中被当作一个个单独的命令直接执行。于是，XDC 后面输入的约束在有冲突的情况下会覆盖之前输入的约束。另外，XDC 中的约束是读一条执行一条，所以先后顺序很重要，一般推荐把时序约束放在前面，而把物理位置约束放在后面。一个 Vivado 工程中可以添加多个 XDC 文件，这样的好处是便于管理组织。因为 XDC 约束条目是有顺序要求的，所以 XDC 文件也是有先后顺序的，可以在 GUI 界面通过鼠标拖拽，来调整 XDC 文件的顺序。

在 XDC 文件中，每个完整的约束指令不应当跨行，必须在一行之内表达完毕。同样，在一行之内也只允许存在一个约束指令，不可以把多个约束放在同一行。

XDC 文件的注释以“#”开头。唯一的限制是，在有效的 XDC 约束指令行末，不可以添加“#”开头的注释，否则 Vivado 会理解错误。这跟我们在其它语言中通常所理解的注释语法不太一样。

常用的 XDC 约束指令如下：

外部时钟输入约束

```
create_clock -period (clock period) -name (clock name) -waveform {(Traise), (Tfall)}  
[get_ports (clock port name)]
```

举例说明

```
create_clock -period 10.000 -name CLK -waveform {0.000 5.000} [get_ports sysclk]
```

已建立的时钟改名

```
create_generated_clock -name (clock name) [get_pins (path)]
```

input/output delay 设置

```
set_input_delay -clock [get_clocks (clock name)] (delay time ns) [all inputs]
```

```
set_output_delay -clock [get_clocks (clock name)] (delay time ns) [all outputs]
```

建立时钟组

```
set_clock_groups -name (group name) -asynchronous -group {(clock name) (clock  
name)}
```

```
set_clock_groups -name (group name) -asynchronous -group [get_clocks (clock  
name)]
```

管脚分配

```
set_property PACKAGE_PIN (pin location) [get_ports (port name)]
```

```
set_property IOSTANDARD (level: LVDS, LVCMOS18, LVCMOS33 etc.)  
[get_ports (port name)]
```

举例说明

```
set_property PACKAGE_PIN T18 [get_ports rst]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

也可以将上面两行合并

```
set_property {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports rst]
```

将非时钟专用管脚用作时钟线

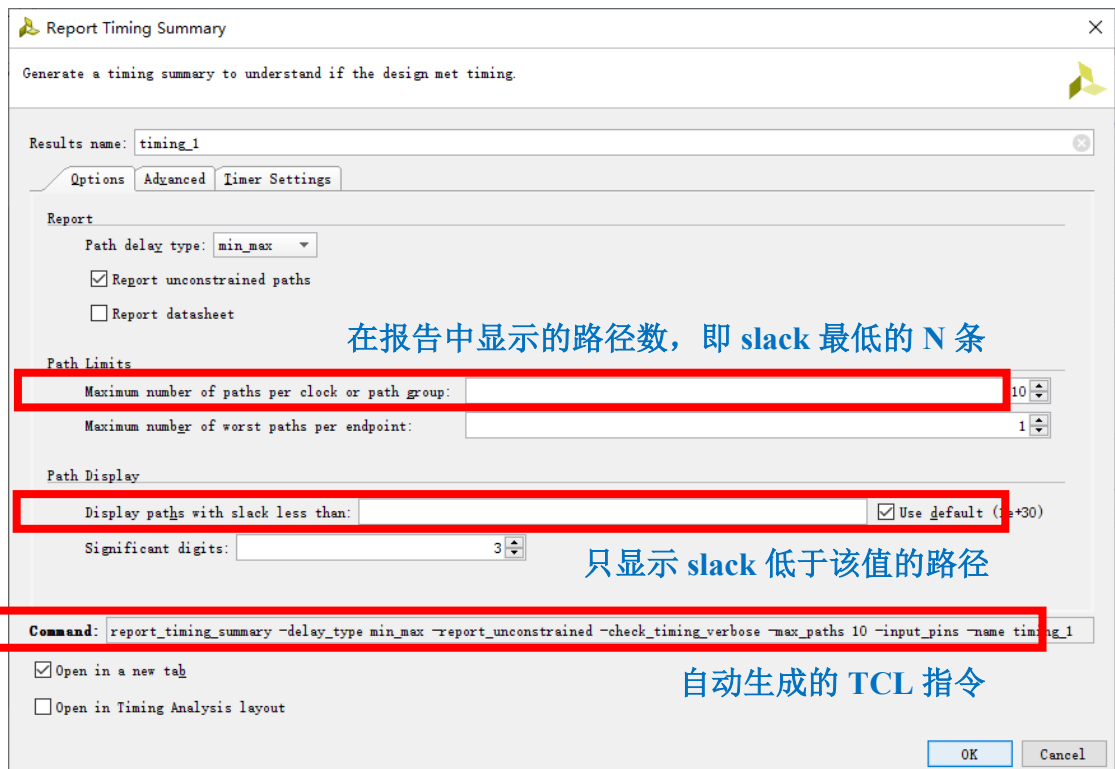
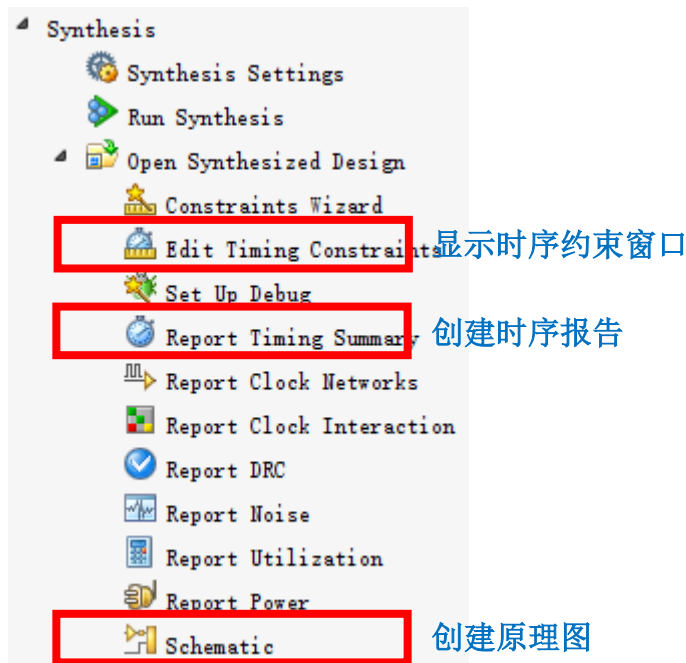
```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets (port_name)]
```

4. 定制时序报告

Vivado 中可以根据需要定制时序分析报告，常用的设置有这些：

1. 报告中显示的路径数。时序报告按照路径的 slack 从低到高显示，所以该设置的含义是显示 slack 最低的 N 条路径。
2. 过滤显示 slack 低于设置值的路径。
3. （图片右侧带框项）显示起点或者终点为某个信号的路径。

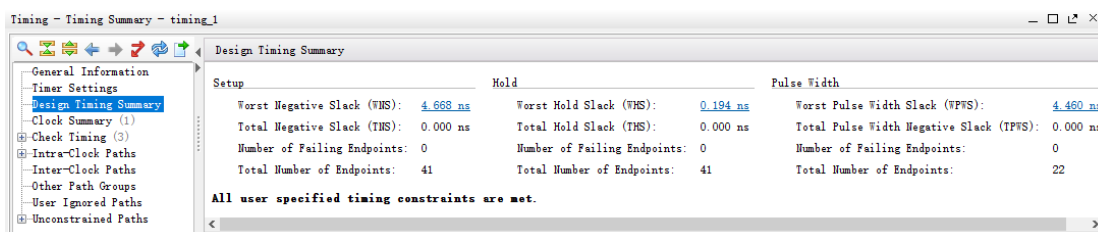
根据用户的设置，Command 一栏会生成相应的 TCL 指令，直接运行这条指令就可以生成一份时序报告，报告参数也由图形界面中的值来定制。



5. 时序报告分析

当关注设计是否产生时序违例，或者关心关键路径的长度时，首先关注 Design Timing Summary。它给出了最直观的信息——是否发生了 Setup/Hold 违例，以及在同步设计中一般不需要关心的 Pulse Width。在此我们以 signinput.v 文件的 sysclk 时钟为例，进行时序分析。结合下图，时序分析报告给出的数据有：

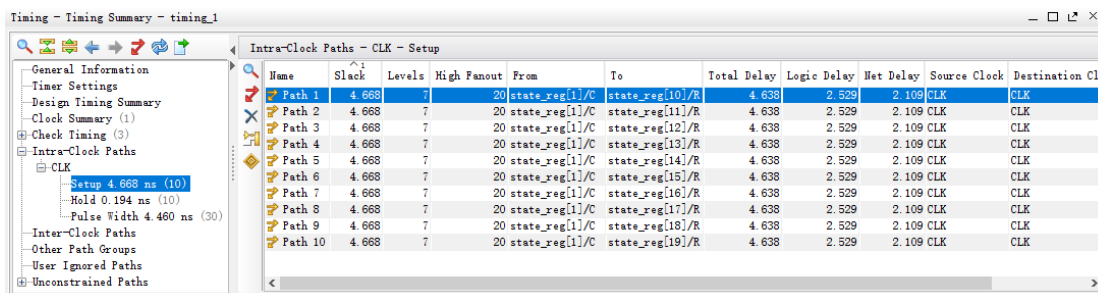
1. 最严重的时序违例有多严重：Worst Negative/Hold Slack (WNS/WHs)
2. 总的时序违例：Total Negative/Hold Slack (TNS/THs)
3. 违例的检查点数量：Number of Failing Endpoint
4. 总检查点数量：Total Number of Endpoint



Setup			Hold			Pulse Width		
Worst Negative Slack (WNS):	4.668 ns		Worst Hold Slack (WHS):	0.194 ns		Worst Pulse Width Slack (WPWS):	4.460 ns	
Total Negative Slack (TNS):	0.000 ns		Total Hold Slack (THS):	0.000 ns		Total Pulse Width Negative Slack (TPWS):	0.000 ns	
Number of Failing Endpoints:	0		Number of Failing Endpoints:	0		Number of Failing Endpoints:	0	
Total Number of Endpoints:	41		Total Number of Endpoints:	41		Total Number of Endpoints:	22	

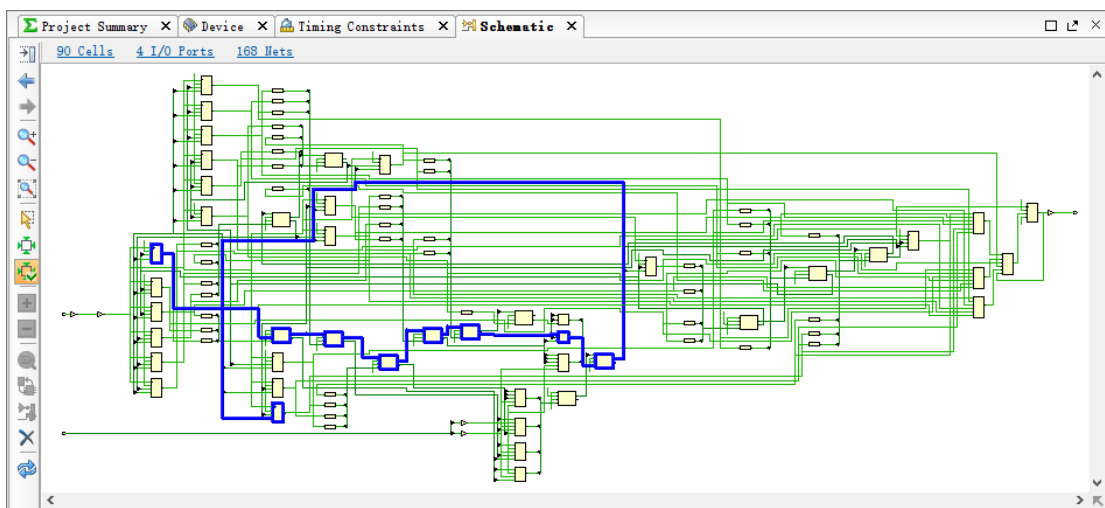
All user specified timing constraints are met.

可以看到，WNS=4.668ns>0、WHS=0.194ns>0，符合要求。未满足要求的信息会以红色标示出。以 WNS 为例，点开“4.668ns”，可以看到哪条路径的时序裕量最小，即为关键路径，如下图所示：

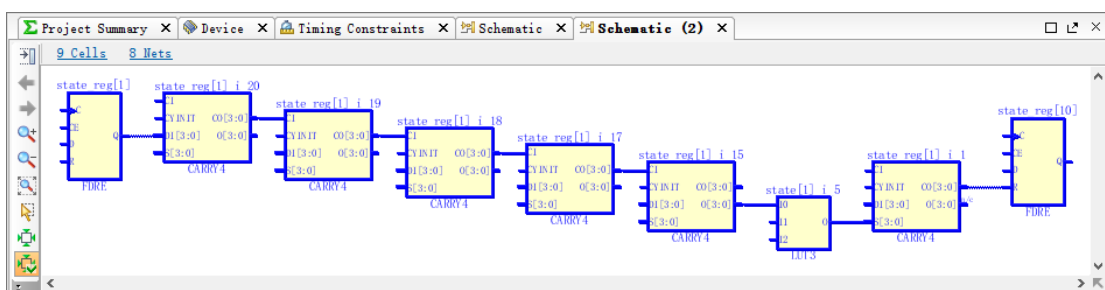


Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Source Clock	Destination Cl.
Path 1	4.668	7	20	state_reg[1]/C	state_reg[10]/R	4.638	2.529	2.109	CLK	CLK
Path 2	4.668	7	20	state_reg[1]/C	state_reg[11]/R	4.638	2.529	2.109	CLK	CLK
Path 3	4.668	7	20	state_reg[1]/C	state_reg[12]/R	4.638	2.529	2.109	CLK	CLK
Path 4	4.668	7	20	state_reg[1]/C	state_reg[13]/R	4.638	2.529	2.109	CLK	CLK
Path 5	4.668	7	20	state_reg[1]/C	state_reg[14]/R	4.638	2.529	2.109	CLK	CLK
Path 6	4.668	7	20	state_reg[1]/C	state_reg[15]/R	4.638	2.529	2.109	CLK	CLK
Path 7	4.668	7	20	state_reg[1]/C	state_reg[16]/R	4.638	2.529	2.109	CLK	CLK
Path 8	4.668	7	20	state_reg[1]/C	state_reg[17]/R	4.638	2.529	2.109	CLK	CLK
Path 9	4.668	7	20	state_reg[1]/C	state_reg[18]/R	4.638	2.529	2.109	CLK	CLK
Path 10	4.668	7	20	state_reg[1]/C	state_reg[19]/R	4.638	2.529	2.109	CLK	CLK

可以看到，这条关键路径从 state[1]的 C 端（时钟端）出发，直到 state[10]的 R 端。如果已经通过选择 Synthesis 下的 Schematic 打开了整体的电路结构图，则这时可以看到对应的路径高亮：



选中 Path 1 再点击 Schematic 可以看到该路径的详细电路图：



展开 Path Properties，得到路径分析报告。报告由四部分组成：Summary、Source Clock Path、Data Path 和 Destination Clock Path。其中，由 Source Clock Path 和 Data Path 可得出 Arrival Time，由 Destination Clock Path 得出 Required Time，二者之差即为时序裕量。以下图为例，展示时序裕量的计算过程：

Source Clock Path				
Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock CLK rise edge)	(x) 0.000	0.000		
net (fo=0)	(x) 0.000	0.000	Site: W5	sysclk
	0.000	0.000		sysclk
			Site: W5	sysclk_IBUF_inst/I
IBUF (Prop ibuf I 0)	(x) 1.458	1.458	Site: W5	sysclk_IBUF_inst/O
net (fo=1, unplaced)	0.800	2.258		sysclk_IBUF
				sysclk_IBUF_BUFInst/I
BUFInst (Prop bufInst I 0)	(x) 0.096	2.354		sysclk_IBUF_BUFInst/O
net (fo=21, unplaced)	0.800	3.154		sysclk_IBUF_BUFInst
				state_reg[1]/C

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
<u>FDRE (Prop fdre C Q)</u>	(r) 0.456	3.610		state_reg[1]/Q
net (fo=4, unplaced)	0.664	4.274		state_reg[1]
<u>CARRY4 (Prop carry4 DI[1] CO[3])</u>	(r) 0.507	4.781		state_reg[1]_i_20/DI[1]
net (fo=1, unplaced)	0.009	4.790		state_reg[1]_i_20/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	4.904		n_0_state_reg[1]_i_20
net (fo=1, unplaced)	0.000	4.904		state_reg[1]_i_19/CI
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.018		state_reg[1]_i_19/CO[3]
net (fo=1, unplaced)	0.000	5.018		n_0_state_reg[1]_i_19
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_18/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_18/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_18
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_17/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_17/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_17
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_15/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_15/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_15
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_13/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_13/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_13
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_11/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_11/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_11
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_9/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_9/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_9
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_7/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_7/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_7
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_5/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_5/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_5
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_3/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_3/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_3
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_1/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_1/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_1
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_0
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_0
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_0
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_0
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0_state_reg[1]_i_0
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		state_reg[1]_i_0/CI
net (fo=1, unplaced)	0.000	5.132		state_reg[1]_i_0/CO[3]
<u>CARRY4 (Prop carry4 CI CO[3])</u>	(r) 0.114	5.132		n_0

可以看到从 FPGA 的时钟输入管脚到第一个触发器的时钟输入用了 3.154ns，触发器 clk-q 的延时是 0.456ns，再加上后续的组合逻辑，到达第二个触发器的 R 端的时间（Arrive Time）是 7.792ns。时钟周期是 10ns 再加上时钟输入到第二个触发器的延时，并考虑时钟的不确定性以及 Setup time，最后得到我们要求 Required Time 是 12.460ns，这个值和 Arrive Time 的差，就是时序裕量 4.668ns。