基于TimeQuest时序优化原理和方法

回顾基于 RTL 逻辑时序优化的基本思路，在关键路径中插入寄存器来优化时序。

分析最坏路径

通过前面对TimeQuest软件的理解，基本上可以找到 关键路径 ，此文章主要对关键路径时序进行优化，使设计达到时序要求，以TFT屏驱动为案例，介绍插入寄存器优化时序的方法；

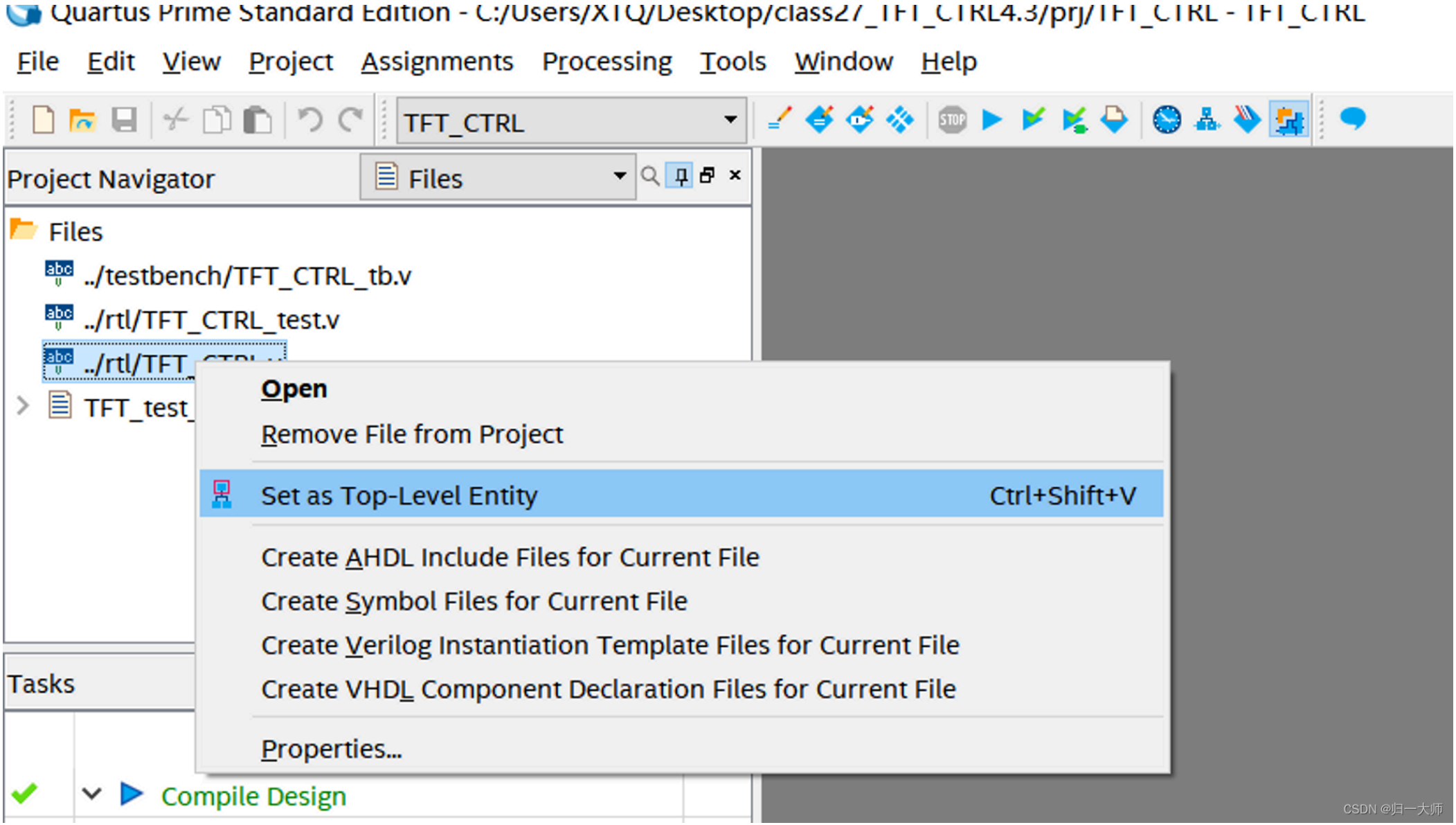


图1 将TFT驱动模块设为顶层模块

将TFT_CTRL设置为顶层之后，ctrl+L对工程进行全编译；

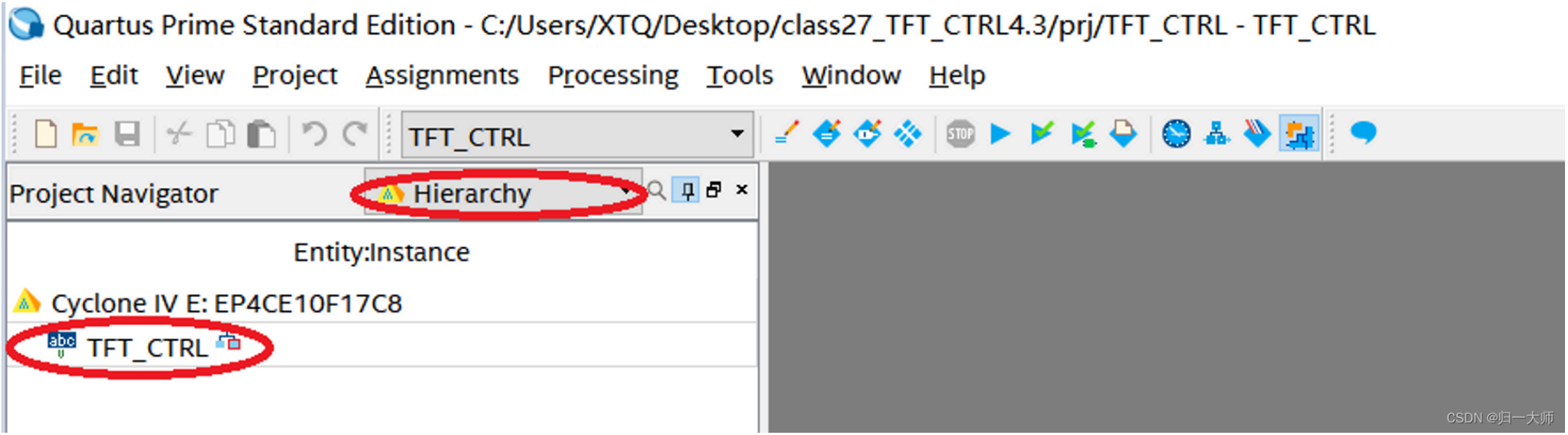


图2 TTF_CTRL设置为顶层模块

全编译之后打开TimeQuest，读取网表，Read SDC之后，查看最高频率为270.12MH，显然无法达到默认时钟1000MHZ，但是这就是设计的最高频率了？不能在提升了？当然可以提升。

Timing Analyzer - C:/Users/XTQ/Desktop/class27_TFT_CTRL4.3/prj/TFT_CTRL - TFT_CTRL

File View Netlist Constraints Reports Script Tools Window Help

Set Operating Conditions

- (●) Slow 1200mV 85C Model
- ( ) Slow 1200mV 0C Model
- ( ) Fast 1200mV 0C Model

Report

- Advanced I/O Timing
- Top Failing Paths
- Clocks Summary
- Fmax Summary
  - Slow 1200mV 85C Model

Tasks

- ✔ Read SDC File
- ✔ Update Timing Netlist
- Reset Design
- Set Operating Conditions...
- Reports

Slow 1200mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	270.12 MHz	250.0 MHz	Clk9M	lim...te)

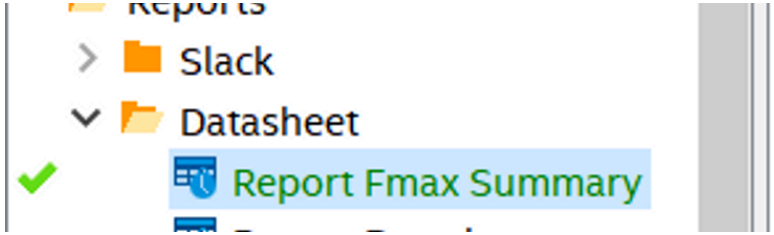


图3 time quest查看最大时钟频率

查看最坏传输路径：

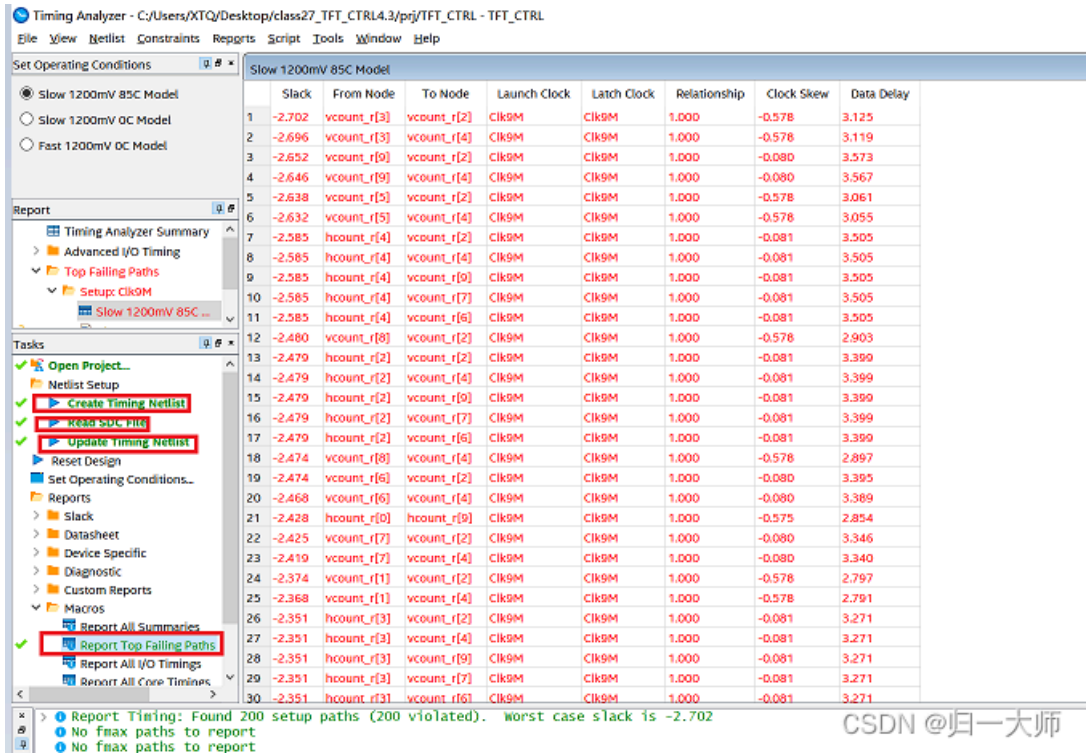


图4 查看最坏路径

可以在报告界面，直接选中 vcounter[2]这个节点，然后鼠标右击，依次选择“Locate-> Locate in Design File”来定位到该路径对应在代码中的位置，如下图所示：

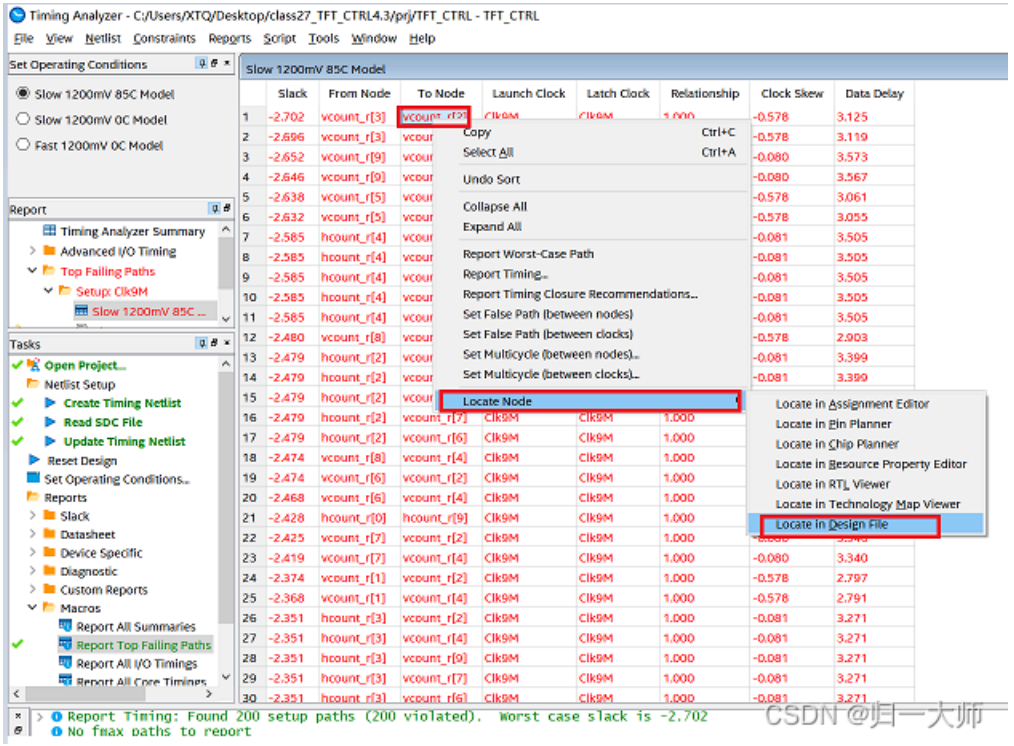
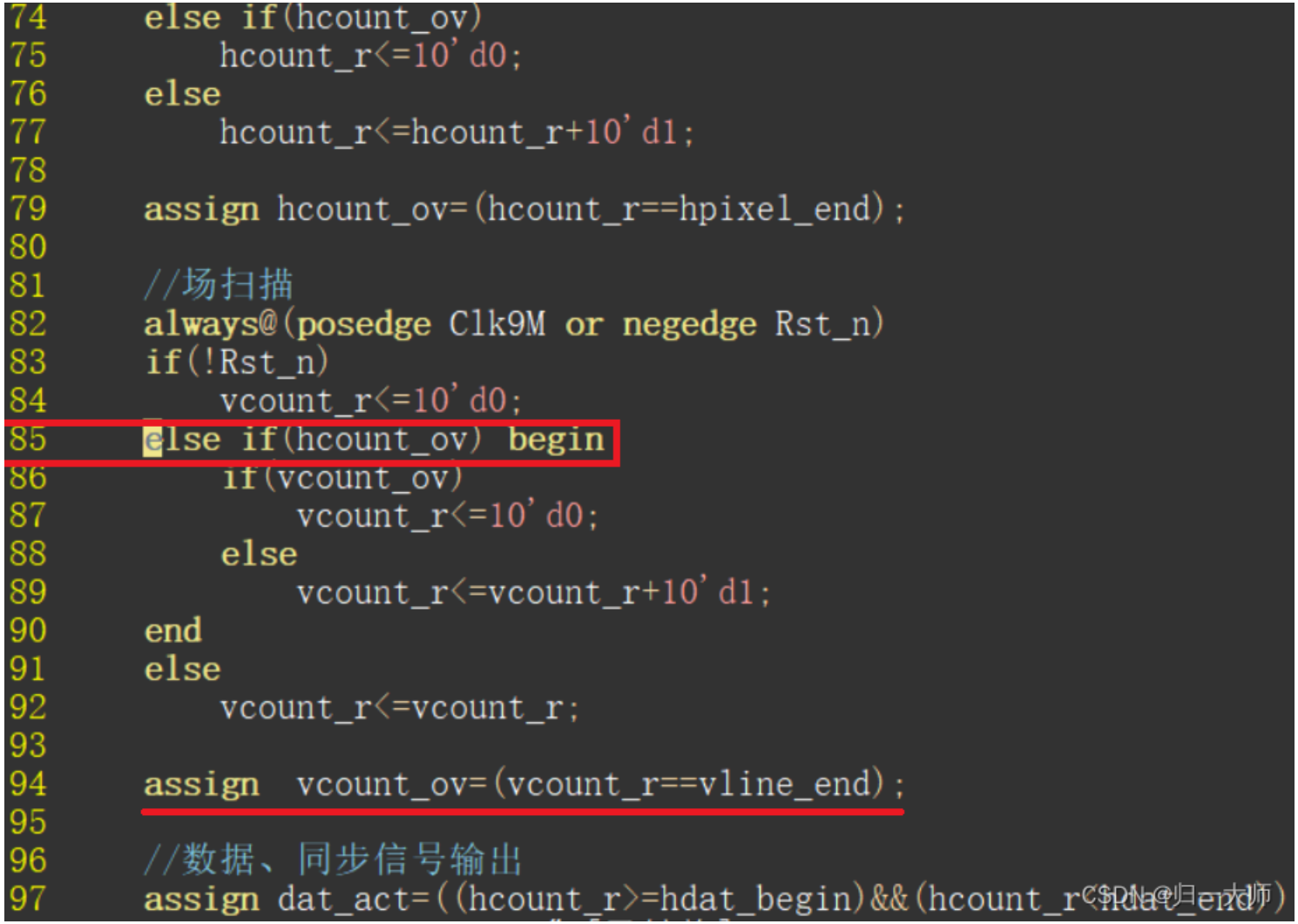


图5 查看最坏路径对应的代码

点击后，软件会自动跳转到代码的第 85 行，如下图所示。在此可以看到，vcount_r的变化是受 hcount_ov 和 vcount_ov 控制的，而 vcount_ov 则是跟随 vcount_r 的值变化的。所以起点和终点就都找到了。



```
74      else if(hcount_ov)
75          hcount_r<=10'd0;
76      else
77          hcount_r<=hcount_r+10'd1;
78
79      assign hcount_ov=(hcount_r==hpixel_end);
80
81      //场扫描
82      always@(posedge Clk9M or negedge Rst_n)
83      if(!Rst_n)
84          vcount_r<=10'd0;
85      else if(hcount_ov) begin
86          if(vcount_ov)
87              vcount_r<=10'd0;
88          else
89              vcount_r<=vcount_r+10'd1;
90      end
91      else
92          vcount_r<=vcount_r;
93
94      assign  vcount_ov=(vcount_r==vline_end);
95
96      //数据、同步信号输出
97      assign dat_act=((hcount_r>=hdat_begin)&&(hcount_r<hdat_end))
```


图6 最坏路径对应代码

在上述路径中，vcount_r 信号要想影响到 vcount_r 的变化，首先是经过一级组合逻辑构成的比较器，在该比较器中，与一个常量（vline_end）做比较，当两者相等时，输出为 1，其他情况下为 0，输出的信号名为 vcount_ov。vcount_ov 的值才来决定 vcount_r 的值是否变化，因此，整个传输路径可以总结为下图：

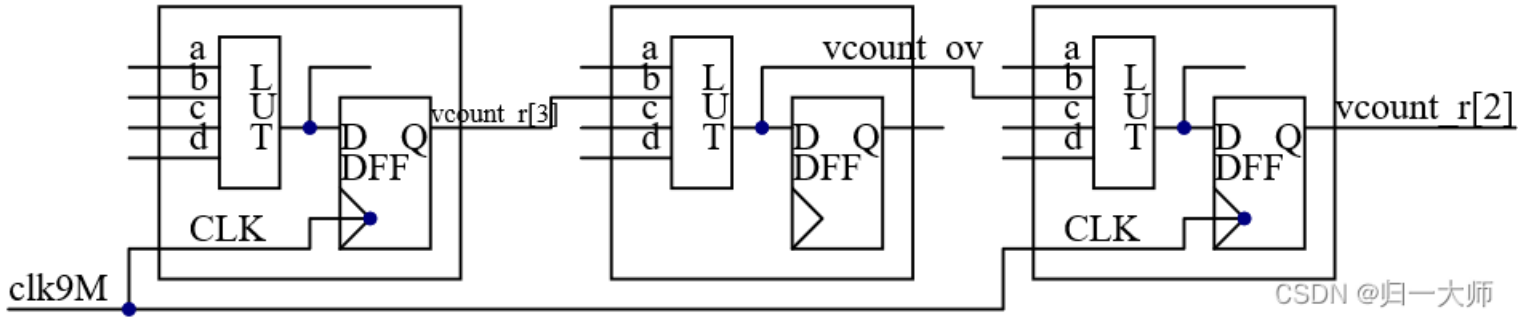


图7 最坏路径传输模型

通过上图可以看到，vcount_r[3]要驱动 vcount_r[2]发生变化。中间一定要经过一级组合逻辑，既然经过了组合逻辑，就一定会引入组合逻辑延迟。而且，整个的路径其实不止这一个组合逻辑，事实上，vcount_ov 有效后，具体 vcount_r[2]是否发生变化，还与 vcount_r 这个寄存器　中的其他位的值相关，所以理论上应该是 vcount_ov 信号还会再进入下一级组合逻辑中参与运算，然后才能最终决定 vcount_r[2]的值是否发生变化。这个组合逻辑在图中被放置在了与 D 触发器 vcount_r[2]同一个 LE 内，这种情况下理论来说延迟是最小的，但是事实上很可能这个组合逻辑并不会与 D 触发器 vcount_r[2]放置在同一个 LE 内，那样的话，延迟就更大了。所以，整个传输路径中至少有 2 级组合逻辑延迟，因为加入了产生 vcount_ov 这个信号的组合逻辑，导致传输延迟变长了。既然这样，只需要将vcount_r[3]到 vcount_r[2]之间的组合逻辑减少，应该就能提升最大运行时钟频率了。

RTL 级路径优化

如何优化路径呢？优化路径的指导思路又是什么呢？其实思路很简单，核心思路就是减少寄存器到寄存器之间的组合逻辑链路。本例中 vcount_r[3]要驱动 vcount_r[2]，中间至少经历两级组合逻辑，如果能够将寄存器到寄存器之间的组合逻辑数量减少一级，是不是就能提升运行的时钟频率了呢？

vcount_r[3]的输出传递到 vcount_r[2]，首先是经历了一级 LUT 实现的查找表，然后查找表的输出再进入另一个查找表，最后才到达 vcount_r[2]，那么，如果能够让第一级查找表输出后，也经过 D 触发器后再送往下一级查找表，那么整个路径就被切为了两段，每段路径都只包含 1 级组合逻辑了，那样的话，传输延迟就会小很多了。如下图所示：

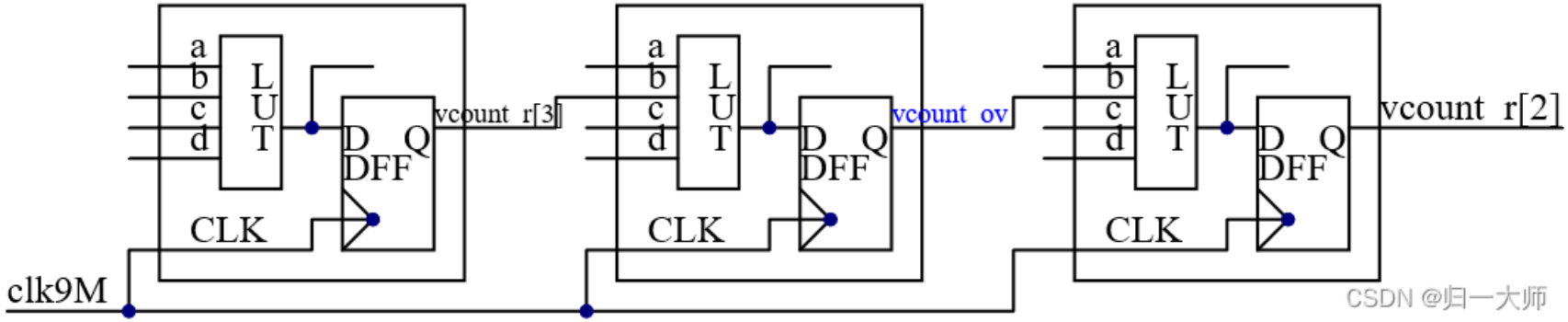


图8 最坏路径优化模型

通过此图可以看到，vcount_ov 不再是直接由 LUT 直接输出，而是 LUT 之后马上进入了该 LE 的 D 触发器中，再由 D 触发器输出。也就是说，此种方式是在原本的时序路径中，插入了一级寄存器，从而将原本较长的组合逻辑链路路径切割为了两段较短的组合逻辑路径。从而让寄存器到寄存器之间的传递组合逻辑延迟更短，提升了系统运行频率。

看上去好像很有道理的样子，那这个操作在原本代码工程中该怎么修改才能实现呢？其实方法非常简单，只需要将 vcount_ov 的产生语句改为时序逻辑即可，当然不要忘记了将该信号的定义也由 wire 改为 reg。具体修改内容为：

1、 代码 49 行，“wire vcount_ov”改为“reg vcount_ov”，如下图所示：

```
45      reg [9:0] hcount_r;      //TFT行扫描计数器
46      reg [9:0] vcount_r;      //TFT场扫描计数器
47      //----------------内部连线定义----------------
48      wire hcount_ov;
49      reg  vcount_ov;
50      wire dat_act;//有效显示区标定
```


图9 修改vcount_ov类型

2、代码 94 行对 vcount_ov 的 assign 赋值语句采用注释的方式屏蔽掉，加上新的时序逻辑描述的代码，如下图所示：

```
90      end
91      else
92          vcount_r<=vcount_r;
93
94      //assign    vcount_ov=(vcount_r==vline_end);
95
96      always  @(posedge Clk9M or negedge Rst_n)begin
97          if(Rst_n==1'b0)begin
98              vcount_ov <= 1'b0;
99          end
100         else begin
101             vcount_ov <= (vcount_r==vline_end);
102         end
103     end
```


图10 修改Vcount_ov处代码

这里将组合电路改为时序电路，vcount_ov 是会延迟一个时钟周期的，为什么不把条件vcount_r ==vline_end 改为vcount_r ==vline_end -1去弥补这一个时钟周期？因为 vcount_r 的变化不是每个时钟都有可能的，只有在 hcount_r 每计满一次才会变化一次，所以这一个时钟周期的延迟没有影响。

那么，上述操作真的就能提升系统性能吗？能提升多少呢？这个嘛，就可以通过修改设计后重新编译，再对设计进行时序分析来知晓了。全编译之后再看一下，结果如下所示：

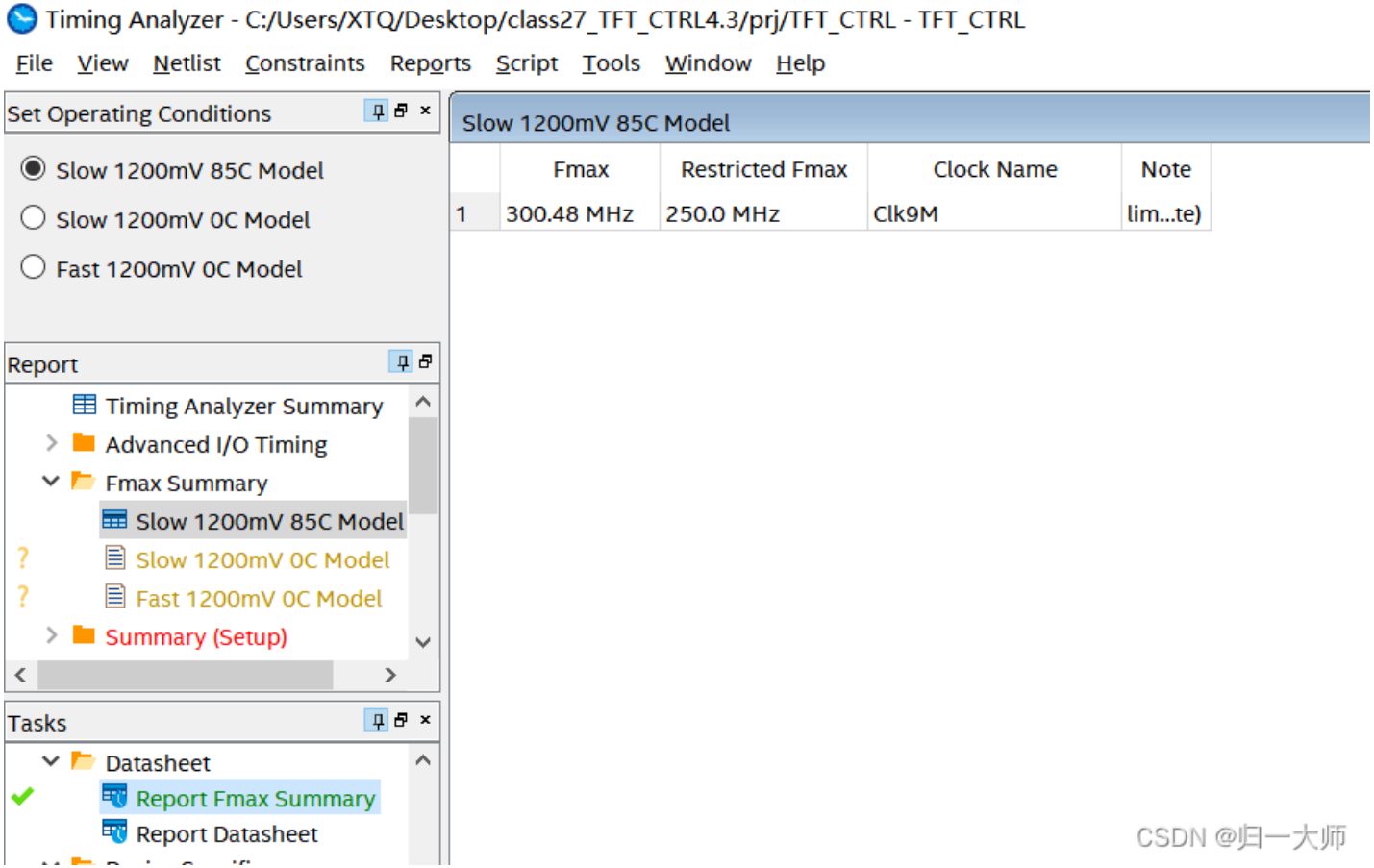


图11 优化结果

300.48MHZ？？？哈哈，一顿操作猛如虎，之前是270.12MHZ，现在是 300.48MHZ，提升了近 30MHzZ了。

首先来说，经过这一波操作，确实没有明显提升最大运行主频，但是这并不代表此方法错了，或者说此方法是没用的，只能说操作还没完。时序分析和约束的过程是一个“约束—>分析—>再约束/修改—>再分析”的往复循环的过程，一次操作只能解决部分问题，当执行了修改之后，也许之前的关键路径解决了，但是马上又会有新的路径成为关键路径，需要再对新的关键路径进行分析，直到最后满足设计需求或者再也无法优化。既然如此，那就继续分析吧，看看经过修改之后，成为新的影响系统运行时钟频率的关键路径是哪个。

通过查看 Worst-Case Timing Paths 下面的 Setup ‘Clk9M’选项，可以看到，此次 vcount_r到vcount_r的这条路径，已经没有提示时序余量为负了，甚至都没有出现在Worst-Case Timing Paths 里面，那么这里，我想留个疑问在这里，供有心的童鞋去思考：此时，hcount_r 到 vcount_r的余量是多少，该怎么看，或者，问个更意外的问题，该条路径是否还在？本笔记不对该问题作答，仅供有心的童鞋去思考。

此次 vcount_r 到 vcount_r 的这条路径，已经没有提示时序余量为负了，时序余量最小的路径是hcount_r[4]到 vcount_r[3]，该路径时序余量为-2.328。

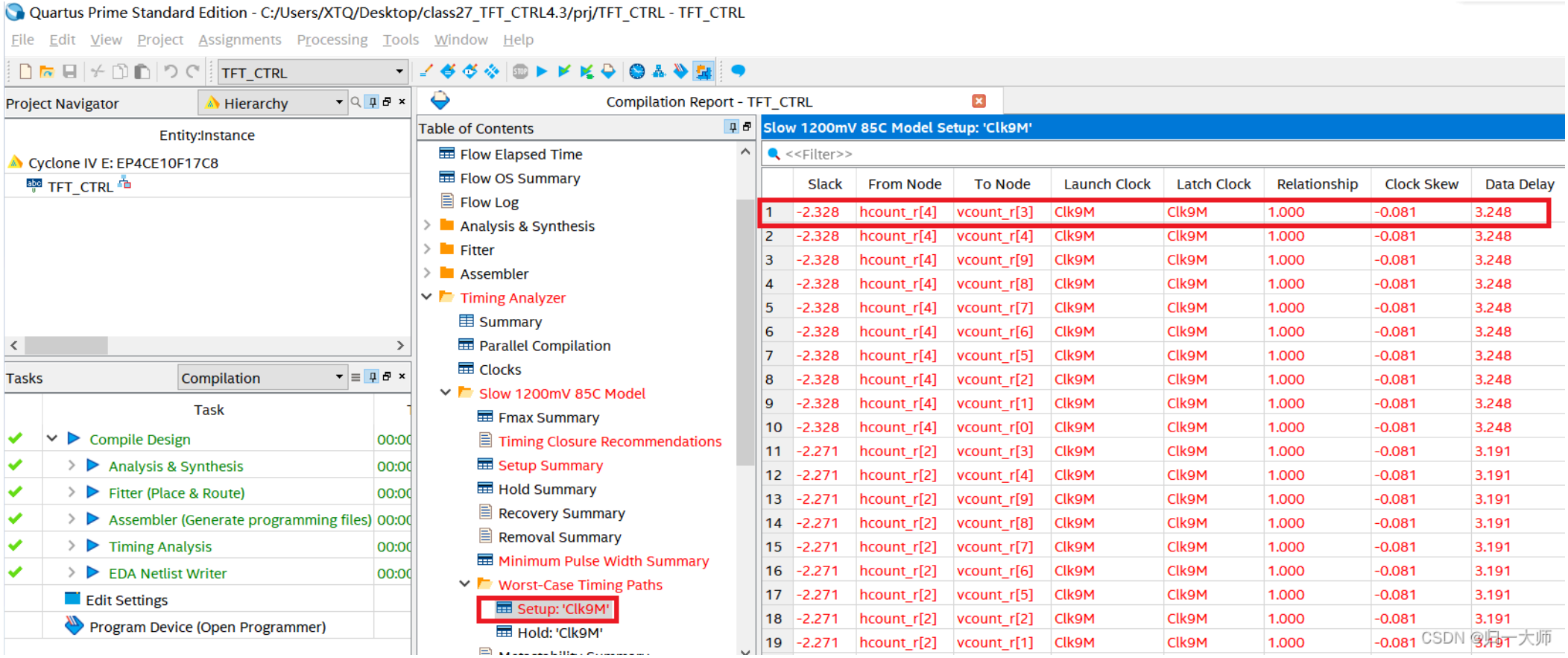


图12 优化后的最坏路径如上图

定位到该路径相关的代码位置，经过分析发现，vcount_r 的计数条件除了和 vcount_ov相关以外，还和 hcount_ov 相关，而 hcount_ov 的产生方法和 vcount_ov 的完全一致，也是使用的组合逻辑直接产生的。

既然这样，那就借鉴前面优化 vcount_ov 的思路，使用同样的方法把 hcount_ov 也优化了，既把 hcount_ov 的产生也改写为时序逻辑。

1、 代码 48行，“wire hcount_ov”改为“reg hcount_ov”，如下图所示：

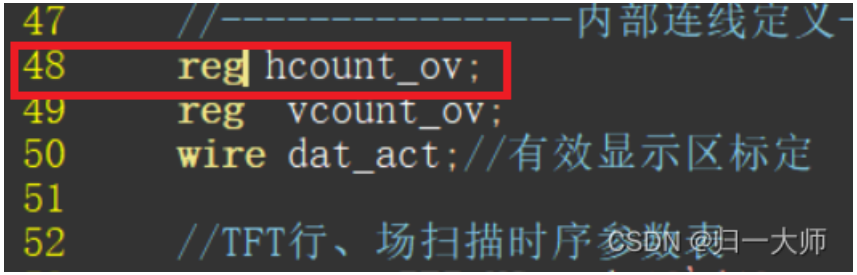


图13 修改hcount_ov信号类型

2、代码 79 行对 hcount_ov 的 assign 赋值语句采用注释的方式屏蔽掉，加上新的时序逻辑描述的代码，如下图所示：

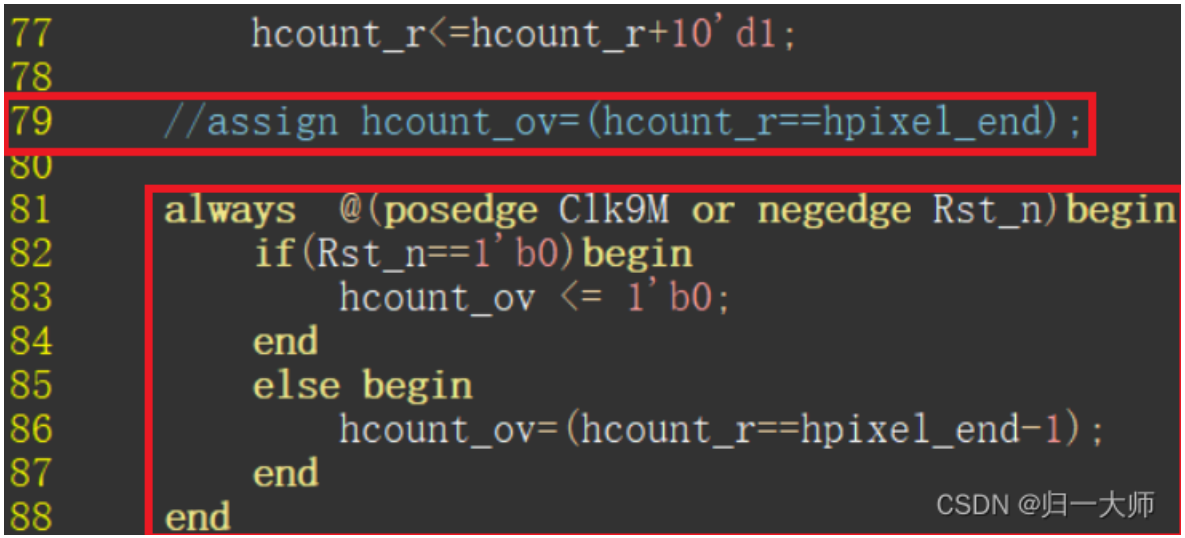
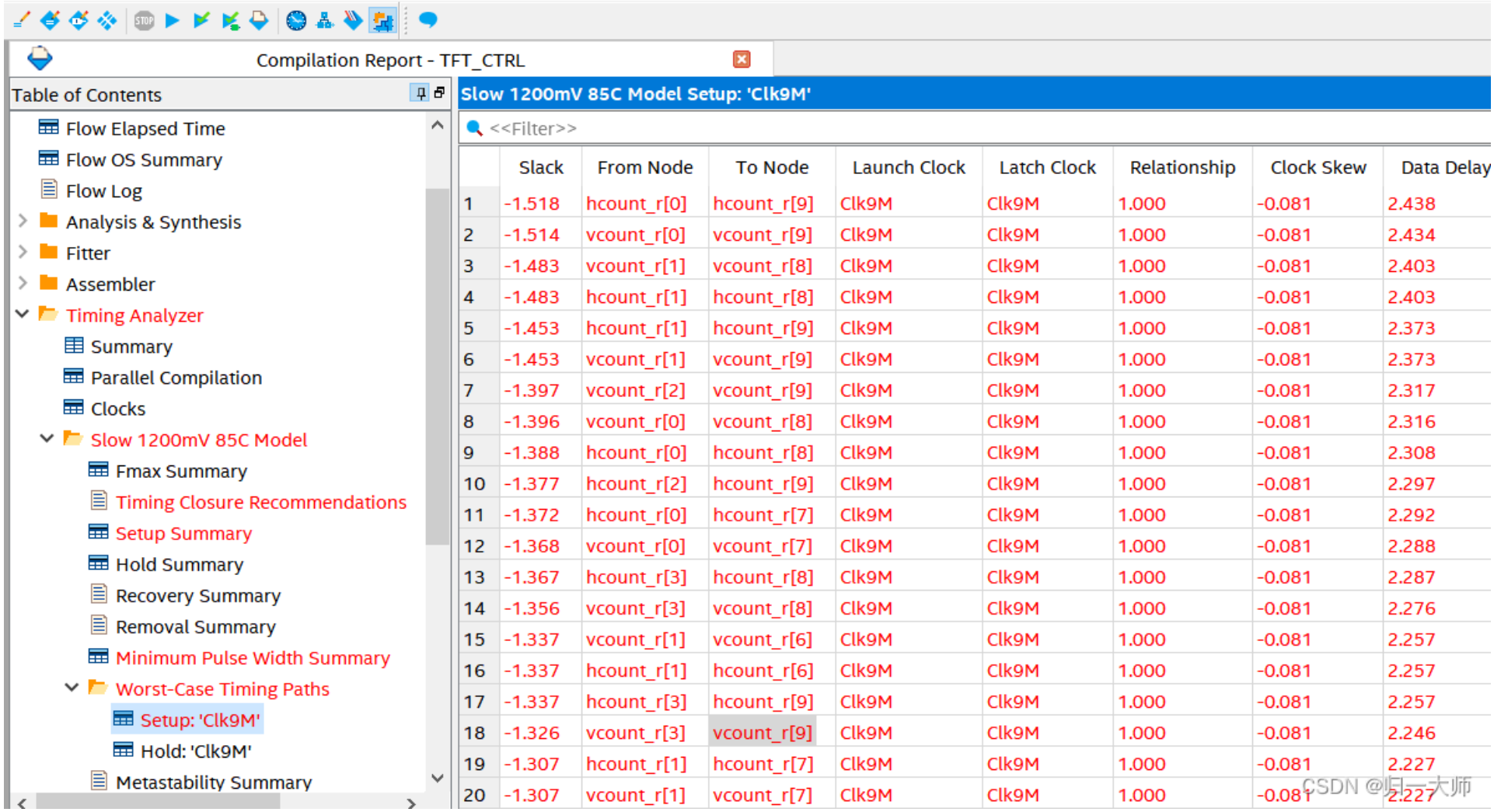


图14 修改hcount_ov处代码

至于图中为啥是 hpixel_end – 1 而不是原来的 hpixel_end ，因为这是寄存器输出，会有一个时钟周期的延迟，所以为了和之前没有修改的时序一致，需要提前一个时钟周期产生该信号。

全编译之后再看一下，结果如下所示：

s Window Help



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-1.518	hcount_r[0]	hcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.438
2	-1.514	vcount_r[0]	vcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.434
3	-1.483	vcount_r[1]	vcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.403
4	-1.483	hcount_r[1]	hcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.403
5	-1.453	hcount_r[1]	hcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.373
6	-1.453	vcount_r[1]	vcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.373
7	-1.397	vcount_r[2]	vcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.317
8	-1.396	vcount_r[0]	vcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.316
9	-1.388	hcount_r[0]	hcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.308
10	-1.377	hcount_r[2]	hcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.297
11	-1.372	hcount_r[0]	hcount_r[7]	Clk9M	Clk9M	1.000	-0.081	2.292
12	-1.368	vcount_r[0]	vcount_r[7]	Clk9M	Clk9M	1.000	-0.081	2.288
13	-1.367	hcount_r[3]	hcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.287
14	-1.356	vcount_r[3]	vcount_r[8]	Clk9M	Clk9M	1.000	-0.081	2.276
15	-1.337	vcount_r[1]	vcount_r[6]	Clk9M	Clk9M	1.000	-0.081	2.257
16	-1.337	hcount_r[1]	hcount_r[6]	Clk9M	Clk9M	1.000	-0.081	2.257
17	-1.337	hcount_r[3]	hcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.257
18	-1.326	vcount_r[3]	vcount_r[9]	Clk9M	Clk9M	1.000	-0.081	2.246
19	-1.307	hcount_r[1]	hcount_r[7]	Clk9M	Clk9M	1.000	-0.081	2.227
20	-1.307	vcount_r[1]	vcount_r[7]	Clk9M	Clk9M	1.000	-0.081	2.227

图15 优化后系统最坏路径

最坏路径的时序余量为-1.518ns，则：Fmax = 1 / (Tclk - Tslack) = 1 / (1 – (-1.518)) = 397.1406MHZ

再看下报告里的 Fmax Summary，也是一样的值：

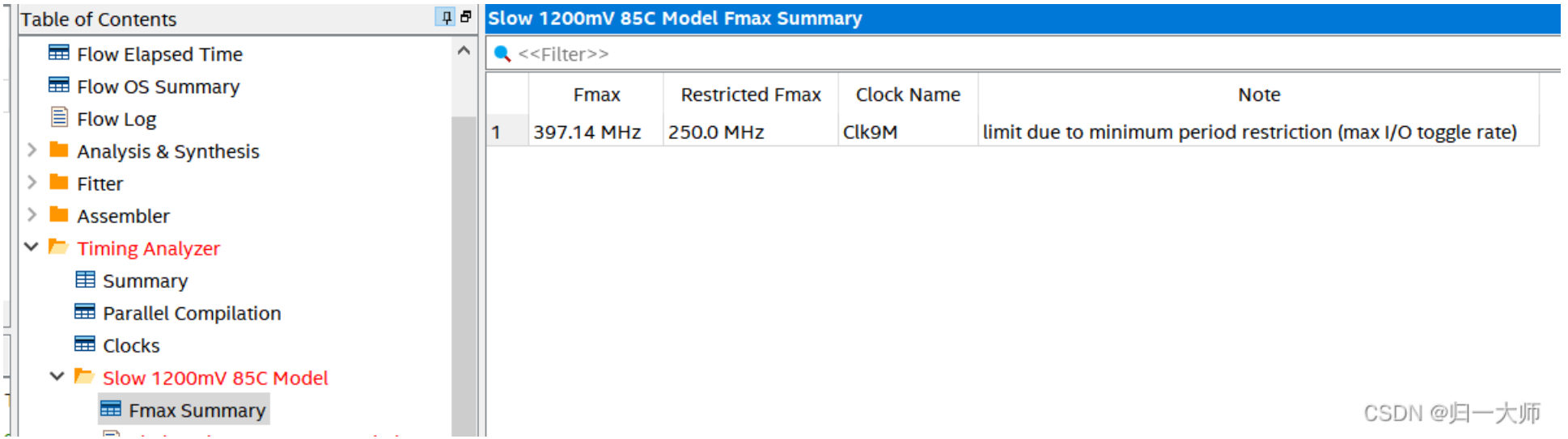


图16 优化后系统最大时钟频率

此时hcount_r 到hcount_r的延迟已经不能够依靠更改RTL代码进行优化了，只能通过更改触发器内部结构才能继续优化，但是这已经不是用户该考虑的问题了；

总结

到此为止，已经优化到了一个比较高的程度了，要再优化，就要从计数器的结构入手了，通过修改计数器的结构来优化，这就很烧脑了。对于当前这个代码，能在 Cyclone IV E 上跑出 393MHz 的频率，相信大家已经很满意了。所以呢关于优化的内容，讲到这里，也就基本差不多了。方法很简单，也就是常说的，插入寄存器大法。

您的支持是我更新的最大动力！将持续更新工程，如果本文对您有帮助，还请多多点赞👍、评论💬和收藏⭐！