

Vivado 下 PLL 实验

黑金动力社区 2018-02-27

1 文档简介

很多初学者看到板上只有一个特定时钟输入的时候都产生疑惑,比如时钟输入 50MHz 或其它频率,那么我要产生一个想要的频率,如 100Mhz、150Mhz 等怎么办?其实在很多 FPGA 芯片内部都集成了 PLL,其他厂商可能不叫 PLL,但是也有类似的功能模块,通过 PLL 可以倍频分频,产生其他很多时钟。本实验通过调用 PLL IP core 来学习 PLL 的使用、vivado 的 IP core 使用方法。

2 实验环境

- Windows 10 64 位
- vivado(vivado2019.1)
- 黑金 FPGA 开发板 (AX7035 开发板、AX7050 开发板)
- 示波器

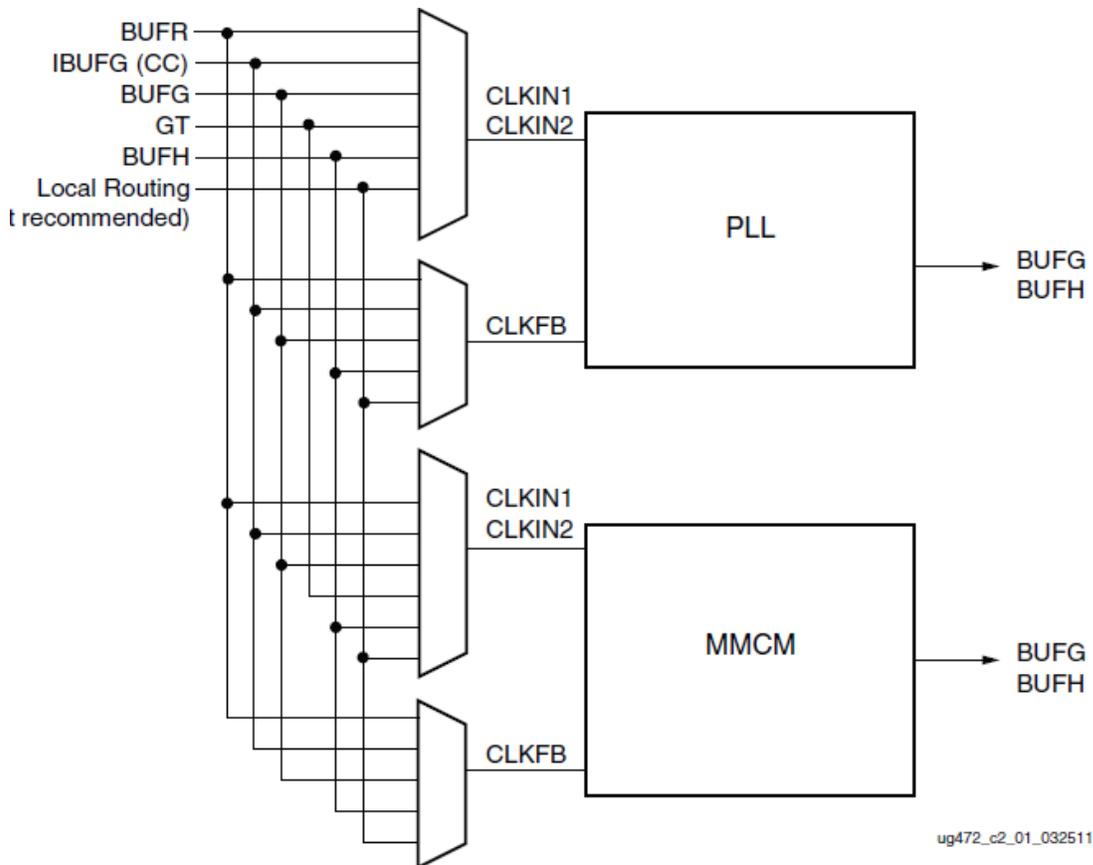
3 实验原理

PLL(phase-locked loop),即锁相环。是 FPGA 中的重要资源。由于一个复杂的 FPGA 系统往往需要多个不同频率,相位的时钟信号。所以,一个 FPGA 芯片中 PLL 的数量是衡量 FPGA 芯片能力的重要指标。FPGA 的设计中,时钟系统的 FPGA 高速的设计极其重要,一个低抖动,低延迟的系统时钟会增加 FPGA 设计的成功率。

本实验将通过使用 PLL, 输出一个方波到开发板上的扩展口 (AX7035 开发板 J10 的 PIN3 脚, AX7050 开发板 J8 的 PIN3 脚), 来给大家演示在 Vivado 软件里使用 PLL 的方法。

7 系列的 FPGA 使用了专用的全局(Global)和区域(Regional)IO 和时钟资源来管理设计中各种的时钟需求。Clock Management Tiles(CMT)提供了时钟合成(Clock frequency synthesis), 倾斜矫正(deskew), 过滤抖动(jitter filtering)功能。

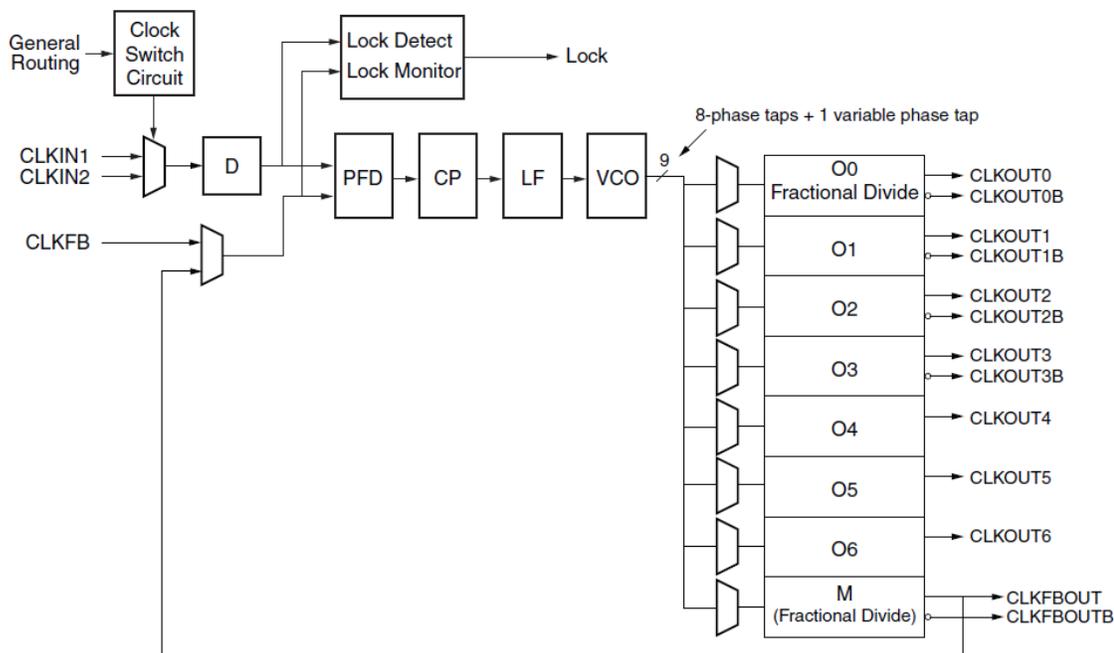
每个 CMTs 包含一个 MMCM(mixed-mode clock manager)和一个 PLL。如下图所示，CMT 的输入可以是 BUFR，IBUFG (CC)，BUFG，GT，BUFH，本地布线（不推荐使用），输出需要接到 BUFG 或者 BUFH 后再使用



➤ 混合模式时钟管理器(MMCM)

MMCM 用于在与给定输入时钟有设定的相位和频率关系的情况下，生成不同的时钟信号。MMCM 提供了广泛而强大的时钟管理功能，

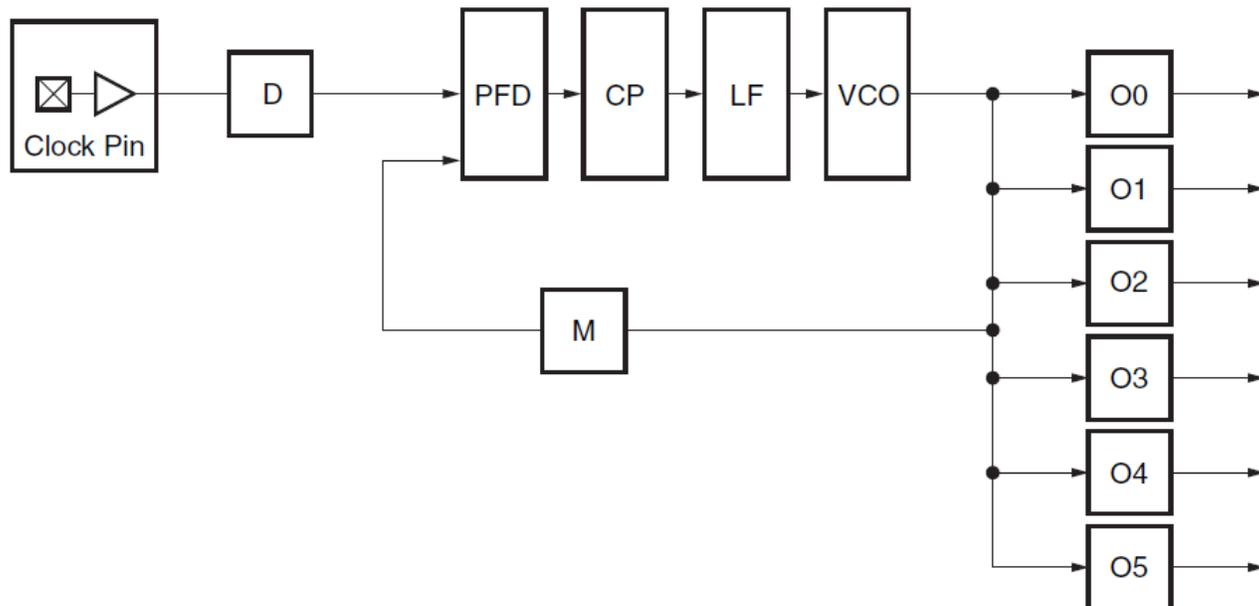
MMCM 内部的功能框图如下图所示:



➤ 数字锁相环(PLL)

锁相环 (PLL) 主要用于频率综合。使用一个 PLL 可以从一个输入时钟信号生成多个时钟信号。

PLL 内部的功能框图如下图所示:



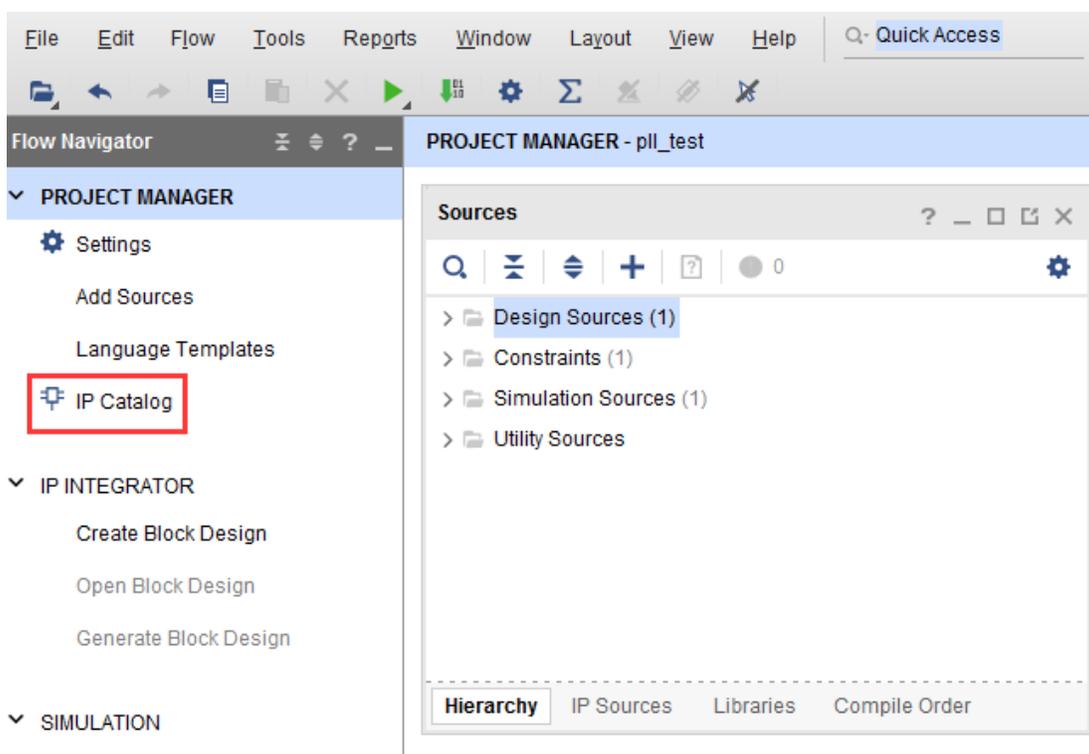
想了解更多的时钟资源, 建议大家看看 Xilinx 提供的文档"7 Series FPGAs Clocking Resources User Guide"。

4 建立工程

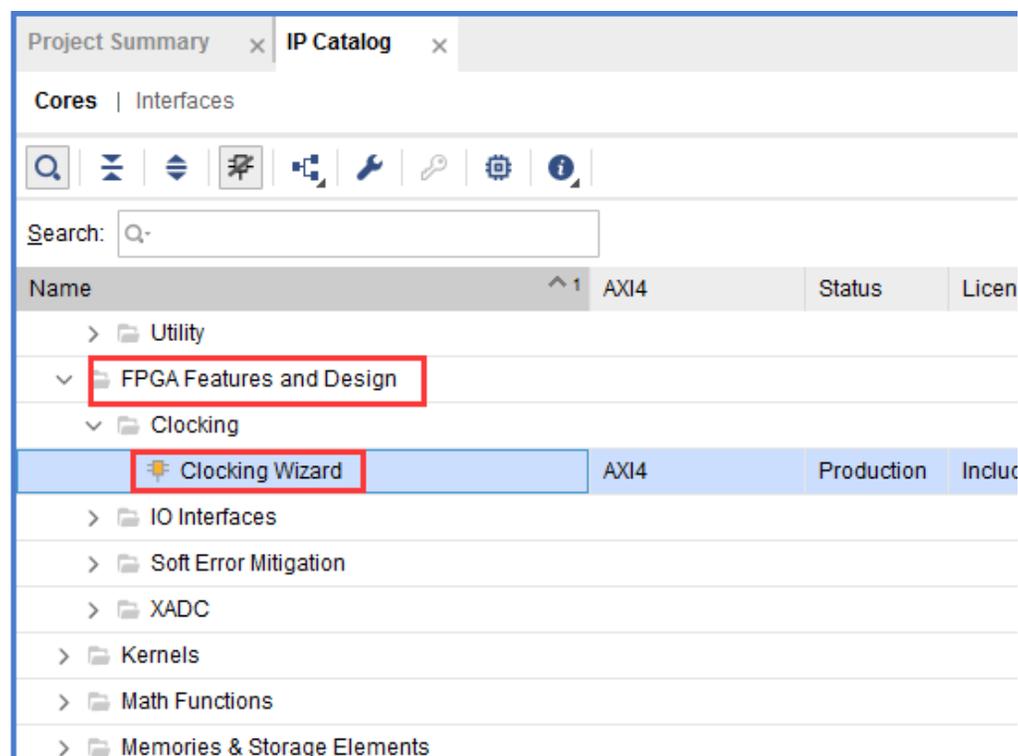
本实验中为大家演示如果调用 Xilinx 提供的 PLL IP 核来产生不同频率的时钟, 并把其中的一个时钟输出到 FPGA 外部 IO 上, 也就是 AX7035 开发板 J10 的 PIN3 脚, AX7050 开发板 J8 的 PIN3 脚。

下面为程序设计的详细步骤。

1. 新建一个 pll_test 的工程, 点击 Project Manager 界面下的 IP Catalog。



2. 再在 IP Catalog 界面里选择 FPGA Features and Design\Clocking 下面的 Clocking Wizard, 双击打开配置界面。



3. 默认这个 Clocking Wizard 的名字为 `clk_wiz_0`, 这里我们不做修改。在第一个界面 Clocking Options 里, 我们选择 PLL 资源, 输入的时钟频率为 50Mhz。

Component Name: clk_wiz_0

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings | Summary

Clock Monitor

Enable Clock Monitoring

Primitive

MMCM PLL

Clocking Features | **Jitter Optimization**

Frequency Synthesis Minimize Power Balanced

Phase Alignment Minimize Output Jitter

Dynamic Reconfig Maximize Input Jitter filtering

Safe Clock Startup

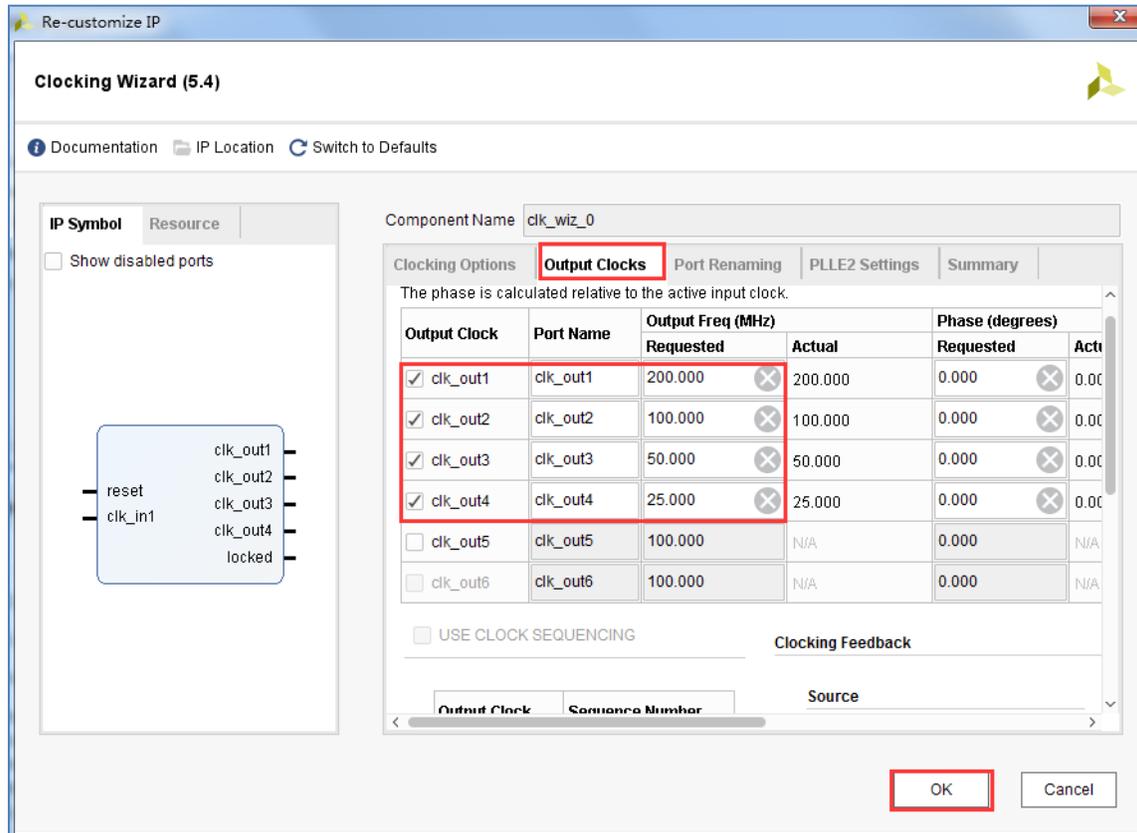
Dynamic Reconfig Interface Options

AXI4Lite DRP Phase Duty Cycle Config Write DRP registers

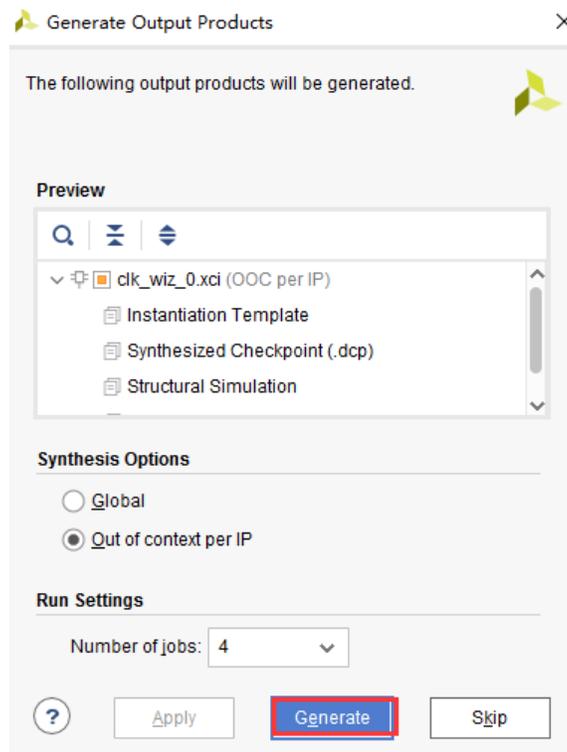
Input Clock Information

Input Clock	Port Name	Input Frequency (MHz)	Jitter Options
<input checked="" type="checkbox"/> Primary	clk_in1	50.000	UI
<input type="checkbox"/> Secondary	clk_in2	100.000	

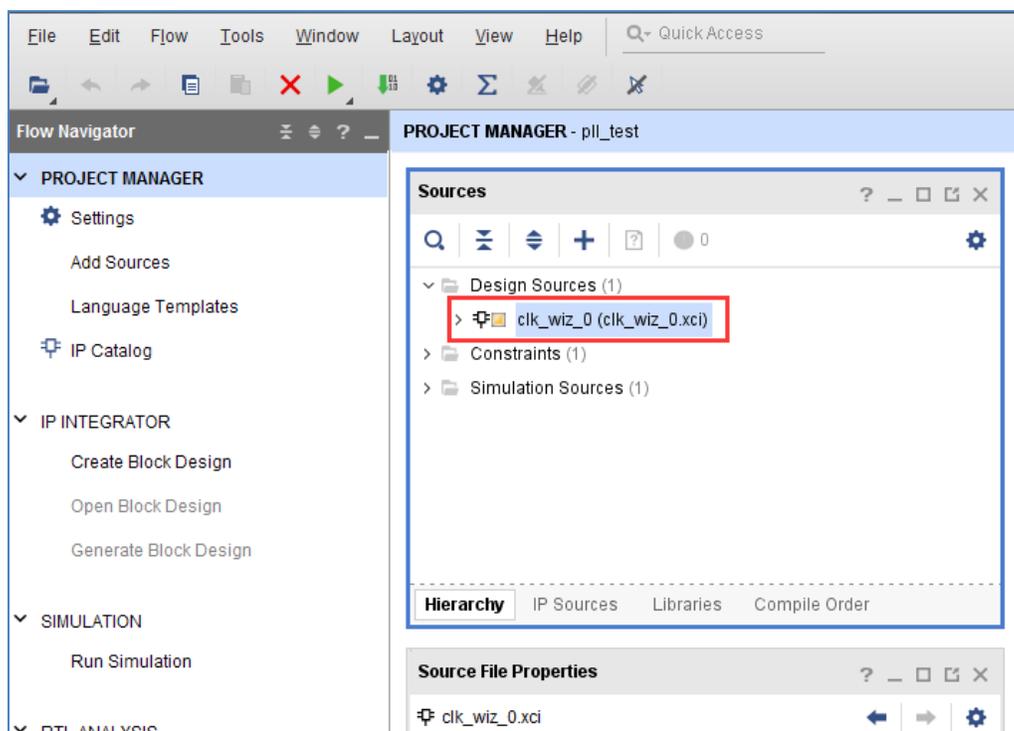
4. 在 Output Clocks 界面里选择 clk_out1~clk_out4 四个时钟的输出，频率分别为 200Mhz, 100Mhz, 50Mhz, 25Mhz。这里还可以设置时钟输出的相位，我们不做设置，保留默认相位, 点击 OK 完成,



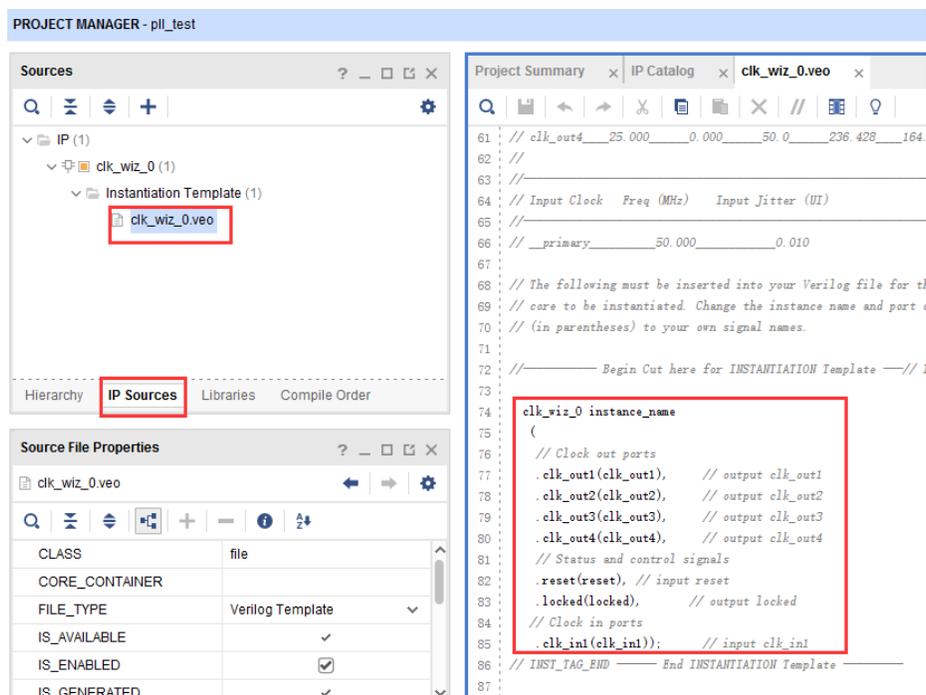
5. 在弹出的对话框中点击 Generate 按钮生成 PLL IP 的设计文件。



6. 这时一个 clk_wiz_0.xci 的 IP 会自动添加到我们的 pll_test 项目中, 用户可以双击它来修改这个 IP 的配置。



选择 IP Sources 这页, 然后双击打开 clk_wiz_0.veo 文件, 这个文件里提供了这个 IP 的实例化模板。我们只需要把框框的中内容拷贝到我们 verilog 程序中, 对 IP 进行实例化。



7. 我们再来编写一个顶层设计文件来实例化这个 PLL IP, 编写 pll_test.v 代码如下。

```

`timescale 1ns / 1ps
module pll_test (
input      sys_clk,           //system clock 50Mhz on board
input      rst_n,           //reset ,low active
output     clk_out           //pll clock output
);

wire       locked;
wire       pll_clk_o;

/*****
          PLL IP calling
*****/
clk_wiz_0 clk_wiz_0_inst
// Clock in ports
.clk_in1   (sys_clk   ),           // IN 50Mhz
// Clock out ports
.clk_out1  (           ),           // OUT 200Mhz
.clk_out2  (           ),           // OUT 100Mhz
.clk_out3  (           ),           // OUT 50Mhz
.clk_out4  (pll_clk_o ),           // OUT 25Mhz
// Status and control signals
.reset     (~rst_n   ),           // RESET IN
.locked    (locked   )
);
/*****
Calling ODDR to make the clock signal output through normal IO
*****/
ODDR #(
.DDR_CLK_EDGE ("SAME_EDGE")
)
ODDR_inst
(
.Q          (clk_out   ),           // 1-bit DDR output data
.C          (pll_clk_o ),           // 1-bit clock input
.CE         (1'b1     ),           // 1-bit clock enable input
.D1         (1'b1     ),           // 1-bit data input (associated with C)
.D2         (1'b0     ),           // 1-bit data input (associated with C)
.R          (1'b0     ),           // 1-bit reset input
.S          (1'b0     )           // 1-bit set input
);
endmodule

```

程序中先用实例化 clk_wiz_0, 把单端 50Mhz 时钟信号输入到 clk_wiz_0 的 clk_in1, 把 clk_out4 的输出付给 pll_clk_o 网络。

注意：例化的目的是在上一级模块中调用例化的模块完成代码功能，在 Verilog 里例化信号的格式如下：模块名必须和要例化的模块名一致，比如程序中的 clk_wiz_0，包括模块信号名也必须一致，比如 clk_in1, clk_out1, clk_out2.....。连接信号为 TOP 程序跟模块之间传递的信号，模块与模块之间的连接信号不能相互冲突，否则会产生编译错误。

```

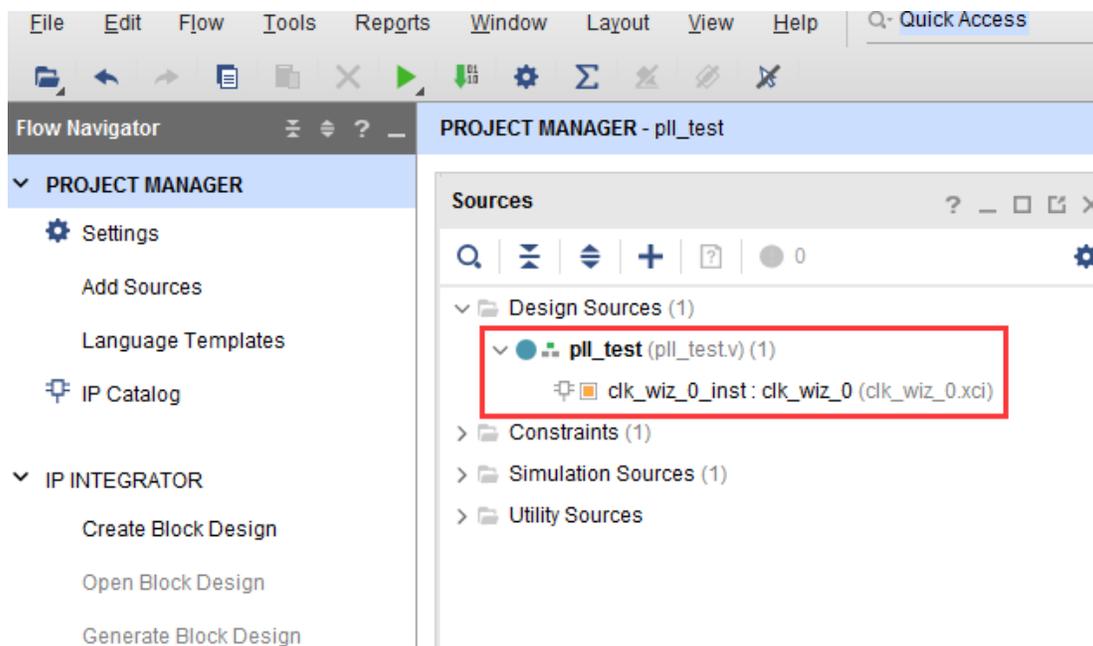
模块名  扩展名
(
.模块信号 1      (连接信号 1) ,
.模块信号 2      (连接信号 2) ,
.模块信号 3      (连接信号 3) ,
.....
.模块信号 N      (连接信号 N)
);

```

另外我们程序中添加了一个 ODDR 原语,使得 clk_wiz_0 的 BUFG 输出的时钟信号能够输出到 FPGA 的普通 IO。因为如果直接从 BUFG 上连接到 OBUF 上,在编译器 map 的过程中就会出现错误,为避免这个错误,通过 ODDR 把两路单端的数据合并到一路上输出,上下沿同时输出数据,上沿输出 a 路下沿输出 b 路;如果两路输入信号一路恒定为 1,一路恒定为 0,那么输出的信号实际上就是输入的时钟信号 pll_clk_o。

另一种方法就是在约束文件中加上一条约束,让编译器忽略时序约束的要求,直接通过普通逻辑资源连接。但这样 Clock 输出的时延和抖动 (Jitter) 都会变差。

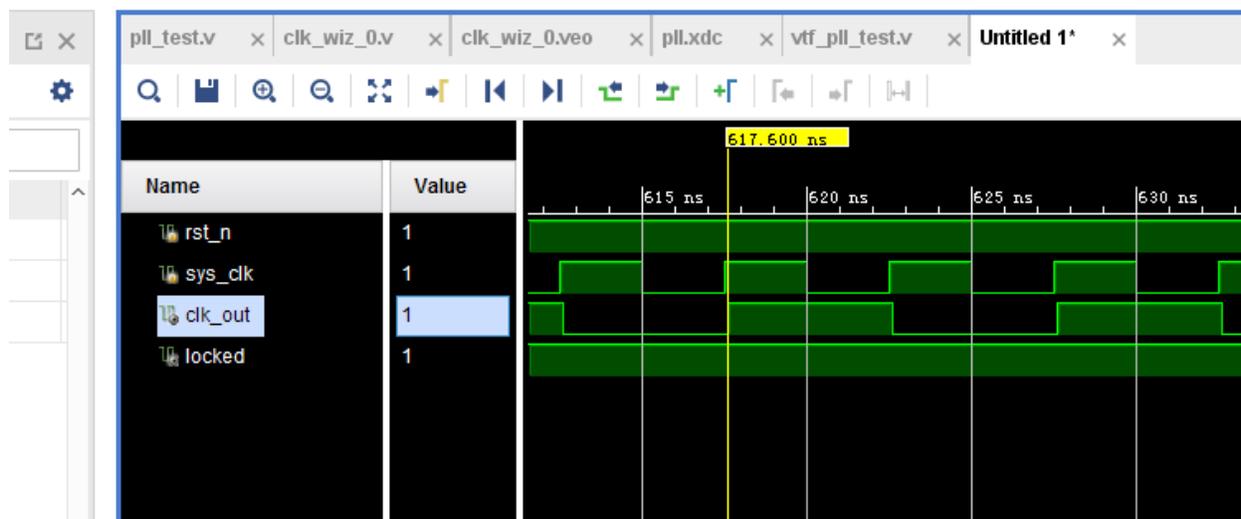
8. 保存工程后, pll_test 自动成为了 top 文件, clk_wiz_0 成为 Pll_test 文件的子模块。



9. 再为工程添加 xdc 管脚约束文件 pll.xdc。

5 仿真

添加一个 vtf_pll_test 仿真文件，运行后 PLL 的 lock 信号会变高，说明 PLL IP 锁相环已经初始化完成。clk_out 有时钟信号输出，输出的频率为输入时钟频率的 1/2, 为 25Mhz。

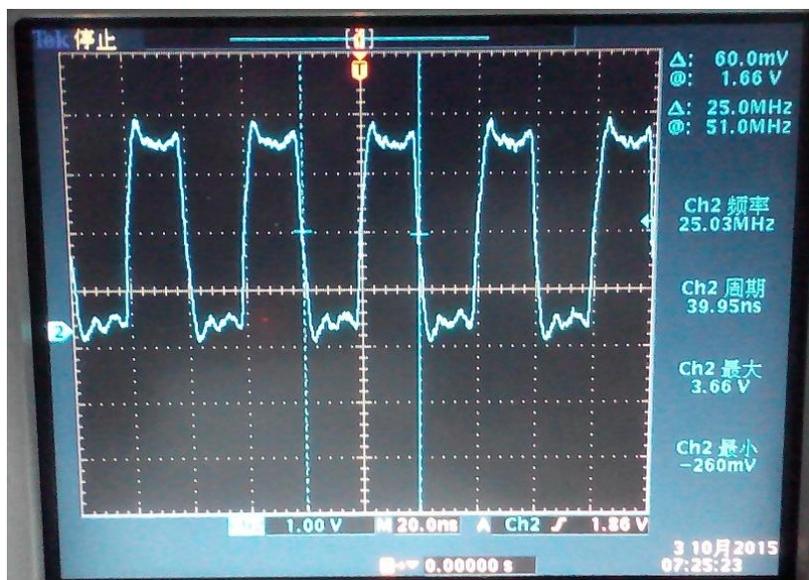


6 测量 PLL 输出波形

编译工程并生成 pll_test.bit 文件，再把 bit 文件下载到 FPGA 中，接下去我们就可以用示波器来测量输出时钟波形了。

用示波器探头的地线连接到开发板上的地 (AX7035 开发板 J10 的 PIN1 , AX7050 开发板 J8 的 PIN1 脚), 信号端连接 AX7035 开发板 J10 的 PIN3 脚或 AX7050 开发板 J8 的 PIN3 脚 (测量的时候需要注意, 避免示波器表头碰到其它管脚而导致电源和地短路)。

这时我们可以在示波器里看到 25Mhz 的时钟波形，波形的幅度为 3.3V, 占空比为 1:1, 波形显示如下图所示：



如果您想输出其它频率的波形，可以修改时钟的输出为 `clk_wiz_0` 的 `clk_out2` 或 `clk_out3` 或 `clk_out4`。也可以修改 `clk_wiz_0` 的 `clk_out4` 为您想要的频率，这里也需要注意一下，因为时钟的输出是通过 PLL 对输入时钟信号的倍频和分频系数来得到的，所以并不是所有的时钟频率都可以用 PLL 能够精确产生的，不过 PLL 也会自动为您计算实际输出接近的时钟频率。

另外需要注意的是，有些用户的示波器的带宽和采样率太低，会导致测量高频时钟信号的时候，高频部分衰减太大，测量波形的幅度会变低。

7 附录

pll_test.v(verilog 代码)

```

`timescale 1ns / 1ps
module pll_test(
input      sys_clk,           //system clock 50Mhz on board
input      rst_n,            //reset ,low active
output     clk_out           //pll clock output
);

wire       locked;
wire       pll_clk_o;

/*****
          PLL IP calling
*****/
clk_wiz_0 clk_wiz_0_inst
  (// Clock in ports
  .clk_in1  (sys_clk      ),           // IN 50Mhz
  // Clock out ports

```

```
.clk_out1 ( ), // OUT 200Mhz
.clk_out2 ( ), // OUT 100Mhz
.clk_out3 ( ), // OUT 50Mhz
.clk_out4 (pll_clk_o ), // OUT 25Mhz
// Status and control signals
.reset (~rst_n ), // RESET IN
.locked (locked )
);
/*****
Calling ODDR to make the clock signal output through normal IO
*****/
ODDR #(
  .DDR_CLK_EDGE ("SAME_EDGE")
)
ODDR inst
(
  .Q (clk_out ), // 1-bit DDR output data
  .C (pll_clk_o ), // 1-bit clock input
  .CE (1'b1 ), // 1-bit clock enable input
  .D1 (1'b1 ), // 1-bit data input (associated with C)
  .D2 (1'b0 ), // 1-bit data input (associated with C)
  .R (1'b0 ), // 1-bit reset input
  .S (1'b0 ) // 1-bit set input
);
endmodule
```