5 | 数字系统综合设计

月色海景与沉船（伊凡·艾瓦佐夫斯基）

思辨尚奥，求索务高，因为我们的归宿在凌霄。

~ · ~

5.1 | 实验目的

综合运用本课程所学习的知识，设计并实现复杂的数字系统。

本实验评分方式较为特殊，所有实验的验收分数将会乘以一个分数系数。在实验5.2.1电子秤与实验5.2.2简单处理器中只需选择一个完成。超出 100% 的部分可折算至考试分数中。具体评分规则见表 5.1。

5.2 | 实验内容

5.2.1 | 电子秤

5.2.1.1 **简易电子秤** 实现一个具有基础功能的电子秤，具体要求如下：

1. 电子秤功能：能计算本次价格与累计价格，并将其显示在数码管上。[1]
   [a] 单次计价：输入物品的重量、单价，显示物品的总价（总价 = 重量 × 单价）。
   [b] 累计计价：
      i. 可以在 Ego1 板上设置累计按键

[1]例如：本次为第 3 次输入累加，单价为 3，质量为 2，历史总价为 20，则数码管可以显示：“AC03 0026”（总输入次数与总价）。按下某个按键后，数码管的显示内容切换为：“0302 0006”（本次的单价、质量与价格）。

ii. 按下累计按键时，记住当前物品的总价（假设当前物品记为物品 1），可采用数码管依次显示以下信息:

AC 次数 应付总价

iii. 继续输入物品 2 的重量、单价，显示物品 2 的总价。按下累计按键，将本次物品 2 的总价累加进之前的用户应付总价中，数码管显示:

AC 次数 总价

iv. 依次购买物品 3、4、……，每个物品后都通过累计按键将本次物品总价累计到应付总价中

v. 进入累计状态的方式可以自由设计。

[c] 退出累计状态：按下清除累计按键，恢复普通状态。

2. 该电子秤具有清零功能, 按下清零键后, 一切恢复如初。数码管应有对应信息输出, 例如 “CLR”。

本实验评分依据系统复杂度、完成程度、系统展示效果、答辩讲述等进行综合评价。

5.2.1.2 UART **电子秤** 基于简单电子秤5.2.1.1，实现一个具有 UART 串口传输功能的电子秤，具体要求如下:

1. 通过 UART 接口实现电脑端和 Ego1 的简单数据传输，要求使用电脑的串口传输软件（例如 XCOM）将重量、单价等数据传输给电子秤，电子秤进行相应的计算、累加、存储等处理，电子秤将需要的反馈结果传输回电脑端（信息格式均可自由设计）。

本实验评分依据系统复杂度、完成程度、系统展示效果、答辩讲述等进行综合评价。

5.2.1.3 **帧解析电子秤** 基于 UART 电子秤5.2.1.2，实现一个具有帧解析功能的电子秤，具体要求如下:

1. 该电子秤具有帧解析功能：**需要设计数据传输的控制协议，规定重量、单价等不同含义的数据的传输格式**，根据该协议电子秤能够解析通过 UART 协议接收的数据桢（识别出重量、单价等不同含义的数据），并对其存储、处理。

帧格式可自行设计，也可参考附录 E。
本实验评分依据系统复杂度、完成程度、系统展示效果、答辩讲述等进行综合评价。

5.2.1.4 **附加功能**

1. 根据我们提供的帧格式，你可以使用最大 4 个字节来存储单价和质量，鼓励进一步实现定点小数的输入与计算功能，例如单价是 12.34，质量是 11.11，那么输出的本次价格为 137.10（四舍五入至 2 位小数）。具体如何实现，答案在你们的创造力中。

2. 该电子秤能在每次输入次数或总价发生改变时（包括清零）通过 UART 向电脑端发送数据，数据包括输入次数、单价、质量、本次价格、总价。电脑端解析该数据，并将其输出在电脑屏幕上。比如可以输出：“AC03 0026 0302 0006”。

本实验鼓励通过整齐的格式给出完整的信息，如:

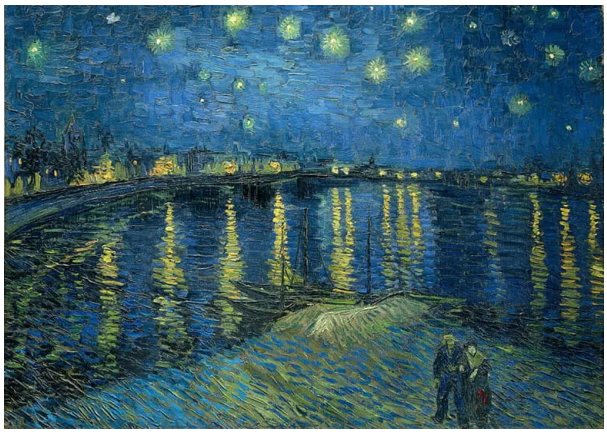
```
Vanity Fair's Balance
price: 1.0$
amount: 2
---accumulate mode begin---
price: 3$
amount: 4
price: 12.34$
amount: 56
---accumulate mode end---
total: xx.xx$
Vanity Fair's Balance
```

具体的输出格式实现，大家可以自由发挥。

3. 设计具有开放性，鼓励设计实现自己的输入输出控制逻辑和显示效果。也可自行实现帧协议，或扩展本实验提供的帧协议。

本实验评分依据系统复杂度、完成程度、系统展示效果、答辩讲述等进行综合评价。

～·～



罗纳河上的星夜（文森特·梵·高）

人类企图攀及星辰的高度，在彩色玻璃和石块上镂刻下自己的事迹。

～·～

5.2.2 | 简单处理器设计

简单的处理器设计主要包括控制器、运算器、数据通路设计和串口通信。

5.2.2.1 **基本功能** 设计并实现一个简单处理器。能够实现 5 种指令 load、move、add、sub 和 show，详细的要求如下：

1. 实现 8 位处理器所要求的全部指令，即寄存器宽度为 8 位（除乘法结果寄存器外）。
2. 处理器中包含 4 个 8 位寄存器存放数据，和一个 16 位的寄存器存放乘法指令的结果。

3. 实现程序计数器 PC（Program counter），用于标志指令的地址（序号）。PC 起始地址为 0，每执行一条指令 PC 加 1。

4. 指令长度为 8 位。

   指令描述：指令长度为 8 位（或扩展为 16 位），采用 4 位操作码，操作数地址都是 2 位

操作	执行的功能
Load Rx, Data	Rx ← Data（数据）
Move Rx,Ry	Rx ← [Ry]
Add Rx,Ry	Rx ← [Rx]+[Ry]
Sub Rx,Ry	Rx ← [Rx] - [Ry]
Show Addr	LED←{Addr,Result[Addr]}

   指令编码格式如下

指令名称	指令格式			
	指令操作码	操作数 1	操作数 2	附加数据段
Load	0000	Rx	00	Data（8 位）
Move	0001	Rx	Ry	无
Add	0010	Rx	Ry	无
Sub	0011	Rx	Ry	无
Show	1111	Addr		无

   [a] Load 加载指令，指令的操作码为 4'b0000，操作数地址分别为 Rx，Ry（不使能），要求加载数据通过外部 Data 信号给出。

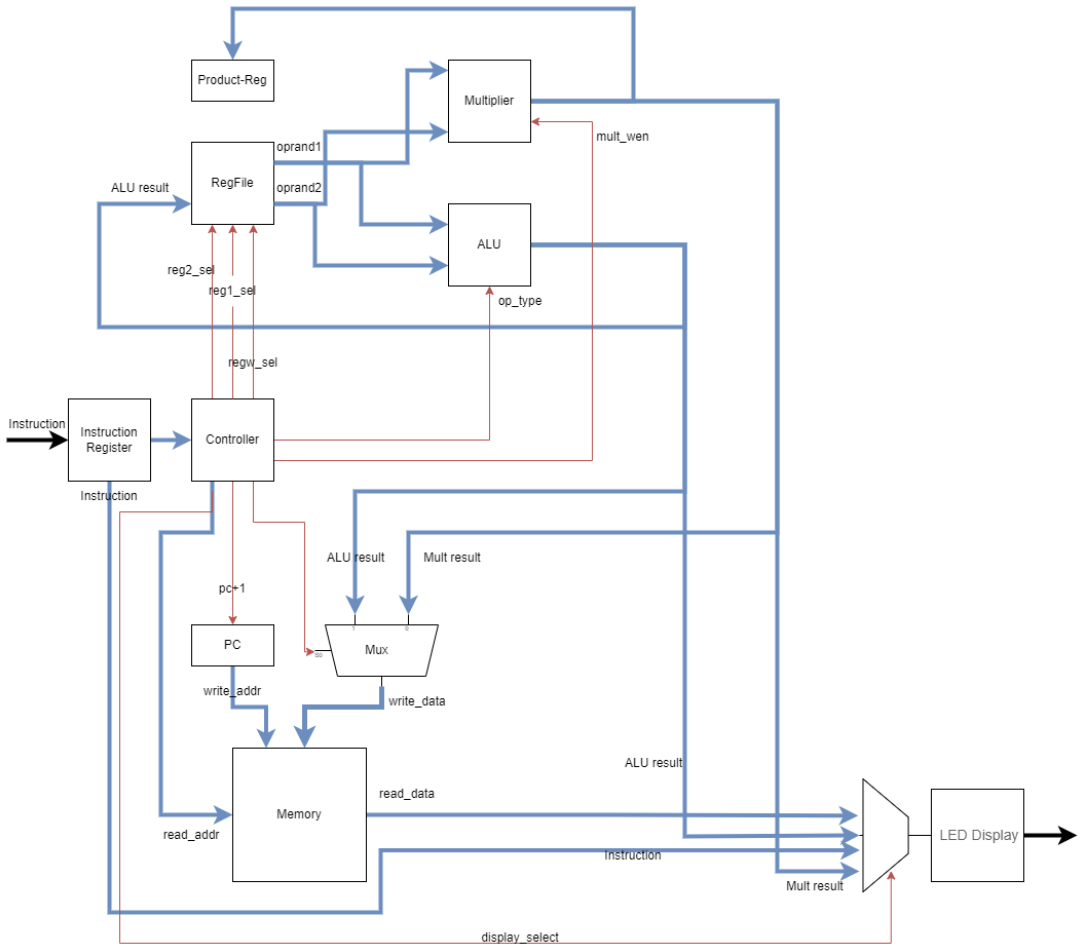
   [b] Move 移动指令，指令的操作码是 4'b0001，操作数地址分别为 Rx，Ry，都为 2 位，将 Ry 指定的寄存器中的值移动到 Rx 指定的寄存器中。

   [c] Add 加法指令，指令的操作码是 4'b0010，操作数地址分别为 Rx,Ry，将 Rx、Ry 指定的寄存器中的数据做加法运算之后，结果保存到 Rx 指定的寄存器中。

   [d] Sub 减法指令，指令的操作码是 4'b0011，操作数地址分别为 Rx,Ry，将 Rx、Ry 指定的寄存器中的数据做减法运算之后，结果保存到 Rx 指定的寄存器中。

   [e] Show 显示命令，指令操作码 4'b1111，原有两寄存器共四位操作数的地址整体作为 Addr。该指令要求能读取在 PC=Addr 时，程序执行的指令结果。高四位数码管显示 PC，低四位数码管显示数据。show 指令不需要将结果写入存储器。

5. 实验可以自行选择实现处理器的方式，以下提供了一种可能的设计，你可以根据自己的需要来修改。图中的红色线表示控制信号，蓝色线表示数据。



5.2.2.2 **乘法扩展** 基于基本功能5.2.2.1，设计并实现一个简单处理器。能够实现 6 种指令 load、move、add、sub、mul 和 show，详细的要求如下：

1. mul 指令要求实现一个模块来进行乘法的计算，在模块内部可以任意选择采用组合逻辑或是时序逻辑的方式实现乘法器。需采用以下几种方式之一实现乘法计算，**不允许使用 IP 核或乘号：**

   [a] https://ustb-806.github.io/DigitalLogic_Info/#/module/unsigned_multiplier 推荐的乘法器

   [b] Booth 译码及华莱士树实现乘法器

   [c] 采用 FPGA 内部 DSP 实现乘法

   乘法指令描述如下：

操作	执行的功能
Mul Rx,Ry	P ← [Rx]*[Ry]

   指令编码格式如下

指令名称	指令格式			
	指令操作码	操作数 1	操作数 2	附加数据段
Mul	0100	Rx	Ry	无

[a] Mul 乘法指令，指令的操作码是 4'b0100，操作数地址分别为 Rx,Ry，将 Rx、Ry 指定的寄存器数据做乘法运算之后，结果保存至 2 倍数据位宽的寄存器 P 中。

5.2.2.3 **扩展功能** 扩展指令中的指令长度为 8 位或 16 位。

1. 将指令存放在存储器当中（允许使用 IP 核），从存储器中读取指令并执行。要求用按键开关为处理器提供时钟，不显示指令内容只选择。
2. 将 UART 集成至 CPU 中，并自定义发送与接收规则，与电脑进行交互。要求 CPU 在启动后应通过串口输出 "USTB LAB6 CPU LAUNCHED." 启动信息。
3. 实现 MovePH，MovePL 指令，指定两个寄存器 Rx,Ry 作为目的寄存器，将乘积 P 放到 Rx 和 Ry 当中。Rx 存放高位，Ry 存放低位。在此基础上计算 $\sum_{k=0}^{5} k!$ 的值

操作	执行的功能
MovePH Rx	Rx←P[15:8]
MovePL Rx	Rx←P[7:0]

4. 实现扩展要求 A，要求基础要求 F 中的存储模块不允许使用 IP 核实现，而是采用开发板提供的 SRAM 作为处理器的存储器。
5. 实现扩展要求 A，在此基础上实现条件跳转指令 BranchEQ, 指令长度为 16 位, 其中四位是指令操作码，其值为 4' b1000，操作数地址为 Rx,Ry，附加数据段是一个四位的立即数 Data.

操作	执行的功能
BranchEQ Rx,Ry,Data Rx	If([Rx]==[Ry])PC←Data

附加指令的编码如下

指令名称	指令格式			
	指令操作码	操作数 1	操作数 2	附加数据段
Load	0000	Rx	00	Data
Move	0001	Rx	Ry	无
Add	0010	Rx	Ry	无
Sub	0011	Rx	Ry	无
Mul	0100	Rx	Ry	无
MovePH	0101	Rx	00	无
MovePL	0110	Rx	00	无
BranchEQ	1000	Rx	Ry	Data
Show	1111	Addr		无

6. 在本实验的基础上，应用前面学习过的各种数字逻辑设计知识，并结合 Ego1 的平台资源（音频接口、VGA 接口、UART 接口、蓝牙接口、通用 I/O 接口等等，可参考用户手册），基于该 CPU 自行搭建一个较为完整的应用示范，更好的掌握复杂数字系统逻辑设计要点。

5.3 | 实验要求

1. 在实验报告中尽量包含：系统设计说明及图、RTL 模块图及必要说明、状态机设计（如果有的话）、测试方案及仿真验证、板级测试结果等。
2. 提交实验报告和实验的完整工程。
3. 本实验需要找助教或老师验收。
4. 所完成的实验可参考其它已有设计，**请遵守学术诚信原则，并引用对应来源。** 在此基础上，不可照搬，需有改进，鼓励原创。

关于 VGA 的参考请见附录F 。

表 5.1: 实验评分表

实验	实验内容	实验要求	分数系数
电子秤 (5.2.1)	简易电子秤 (5.2.1.1)	5.2.1.1-1 5.2.1.1-2	35%
	UART 电子秤 (5.2.1.2)	5.2.1.2-1	50%
	帧解析电子秤 (5.2.1.3)	5.2.1.3-1	60%
	扩展电子秤 (5.2.1.4)	5.2.1.4-1 5.2.1.4-2	70%
简单处理器 (5.2.2)	基本功能 (5.2.2.1)	5.2.2.1-1 5.2.2.1-2 5.2.2.1-3	90%
	乘法扩展 (5.2.2.2)	5.2.2.2-1	100%
	任意一项扩展 (5.2.2.3)	5.2.2.3-1 ∼ 5.2.2.3-6	125%
	任意两项扩展 (5.2.2.3)	5.2.2.3-1 ∼ 5.2.2.3-6	150%
	任意三项扩展 (5.2.2.3)	5.2.2.3-1 ∼ 5.2.2.3-6	175%
	任意四项扩展 (5.2.2.3)	5.2.2.3-1 ∼ 5.2.2.3-6	200%

E | 帧协议及帧解析器模块

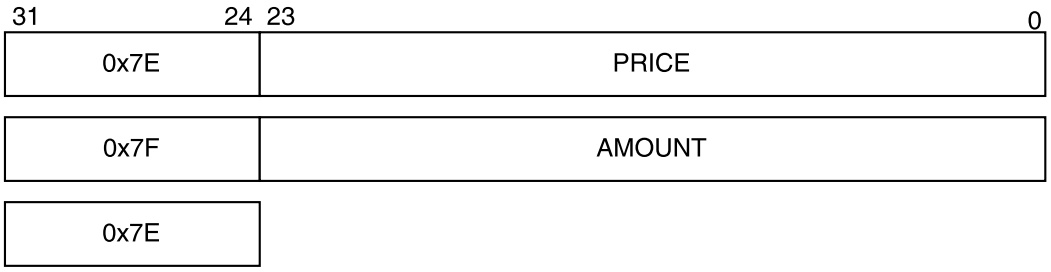


图 E.1: 帧定义

0x7E 是帧开始和结束的标志。当第一个 0x7E 出现，代表一个帧开始，而当第二个 0x7E 出现，代表一个帧结束。在出现一个 0x7E 后，接下来的字节代表 PRICE 字段，而当出现了一个 0x7F，代表 PRICE 字段结束，AMOUNT 字段开始。

在本协议中，如果 PRICE 或 AMOUNT 字段中出现了 0x7E，0x7F 则需要进行转义。对于每一个 0x7E 而言，需要写为 0x7D 0x5E；对于每一个 0x7F 而言，需要写为 0x7D 0x5F。而若想要在 PRICE 或 AMOUNT 字段中输出 0x7D，则需要转为 0x7D 0x5D。

在本模块中，0x7D 0x5F 的转义部分没有完成，你可以选择不使用扩展的帧协议，即使用简化版本；也可以选择自行实现帧协议，或者自行在本模块中添加 0x7D 0x5F 的转义。

接收端将会按照以上规则进行处理，因此在发送时，请注意发送的信息有时需要进行转义。

E.1 | PLL，FIFO IP 核的使用

实验可直接使用 Ego1 板子上提供的 100MHz 时钟信号，也可以按照 UART 手册中建议的 150MHz 时钟信号，如果需要使用 150MHz 时钟作为 UART input clock，则需要使用 Vivado 的 PLL IP 核从 EGO-1 提供了 100MHz 的时钟信号生成 150MHz 时钟信号。

请依照以下步骤进行 IP 核心的例化：

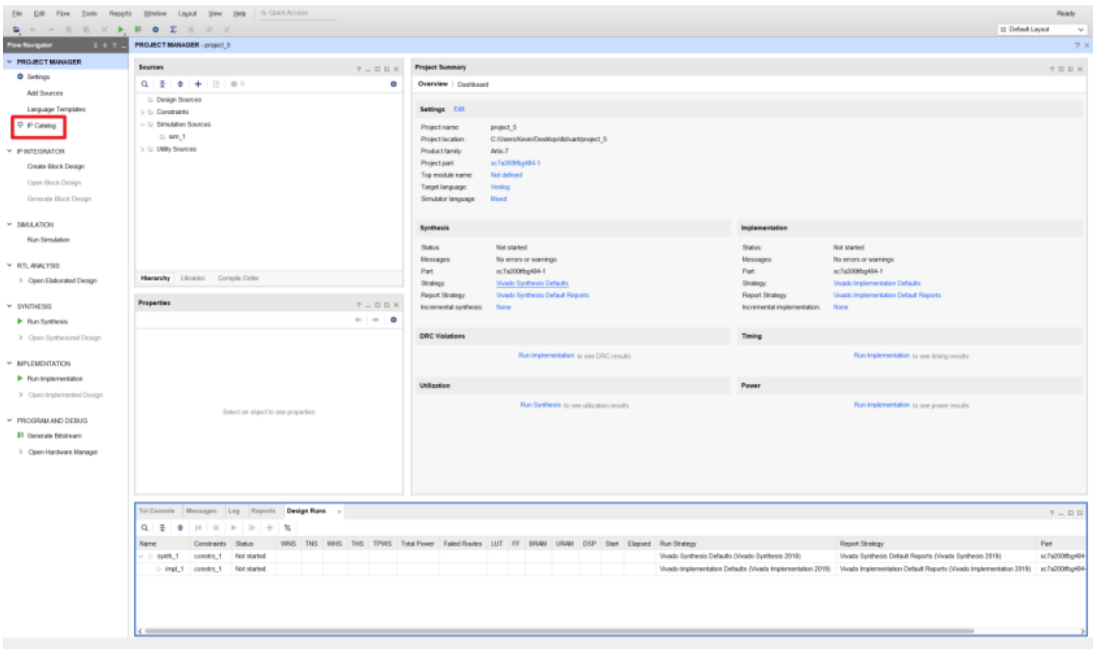


图 E.2: 选择 IP Catalog

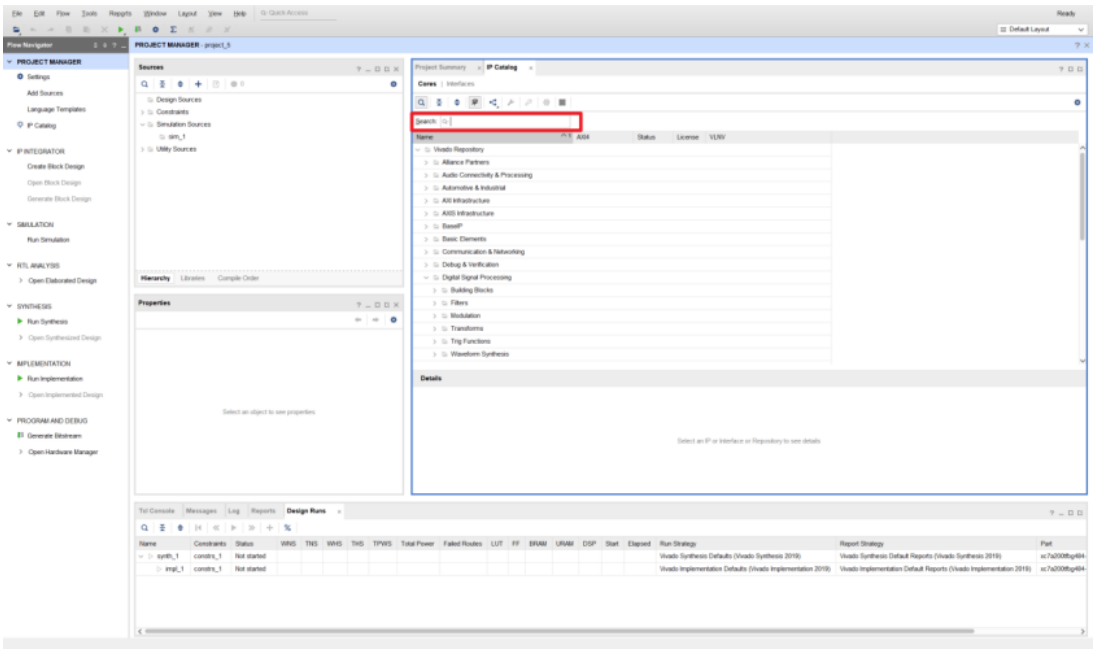


图 E.3: 搜索 IP 核

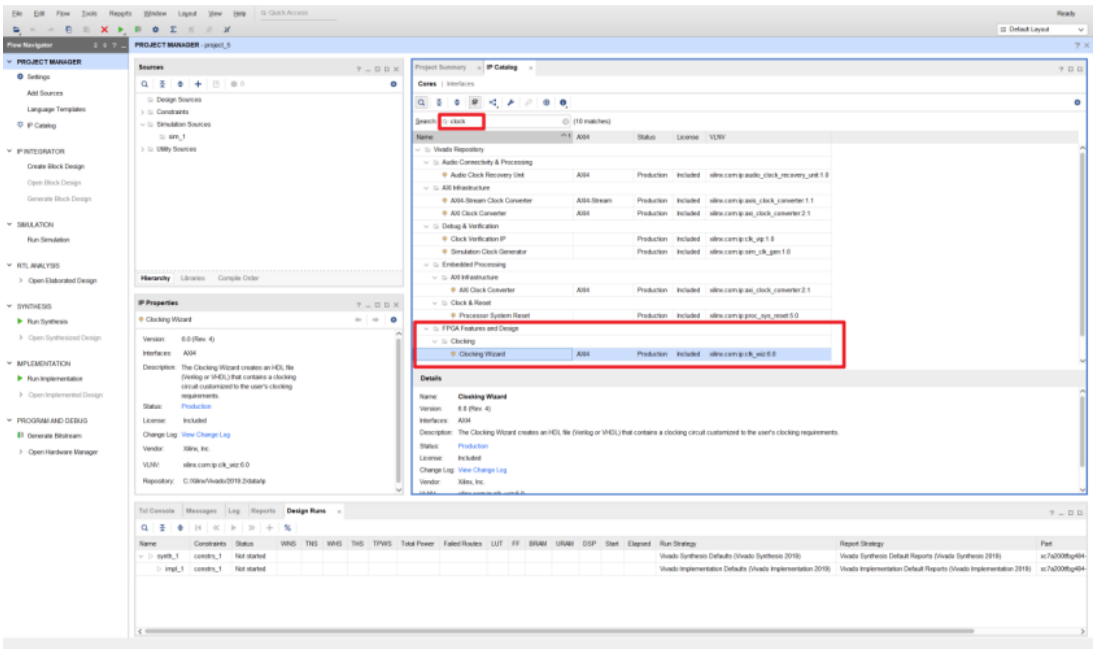


图 E.4: 选择 IP 核

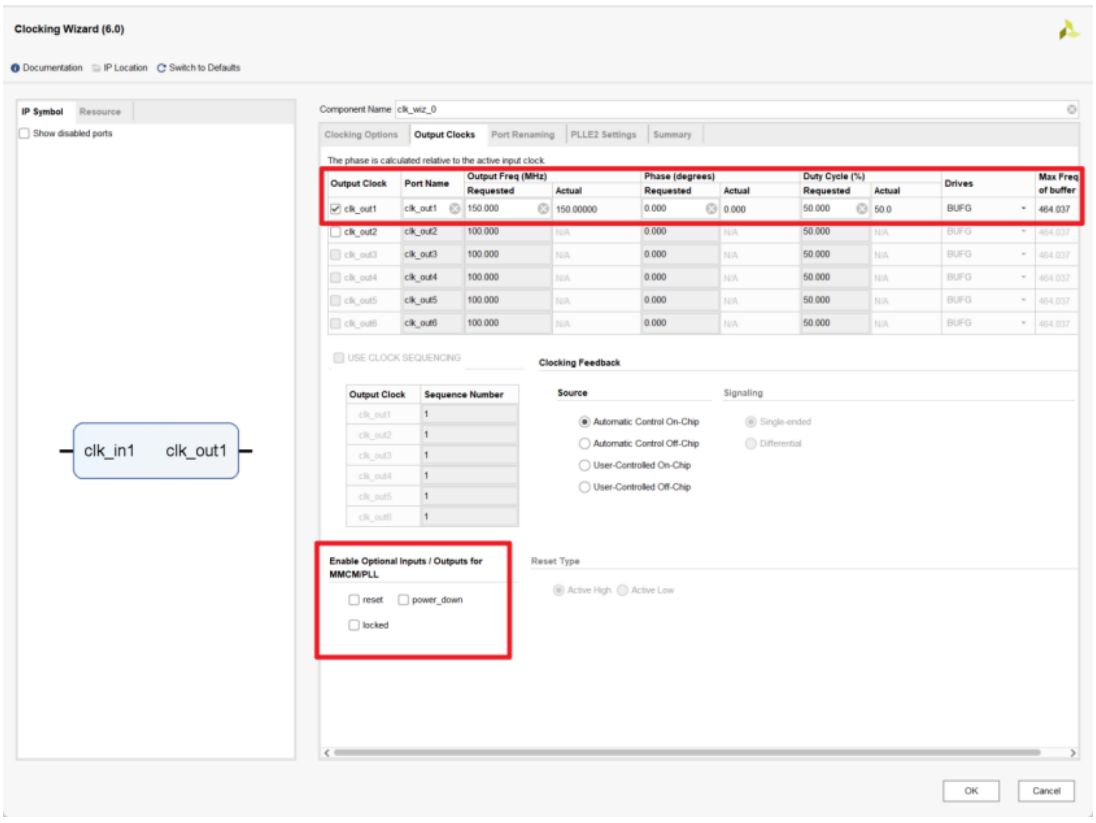


图 E.5: 设置 IP 核

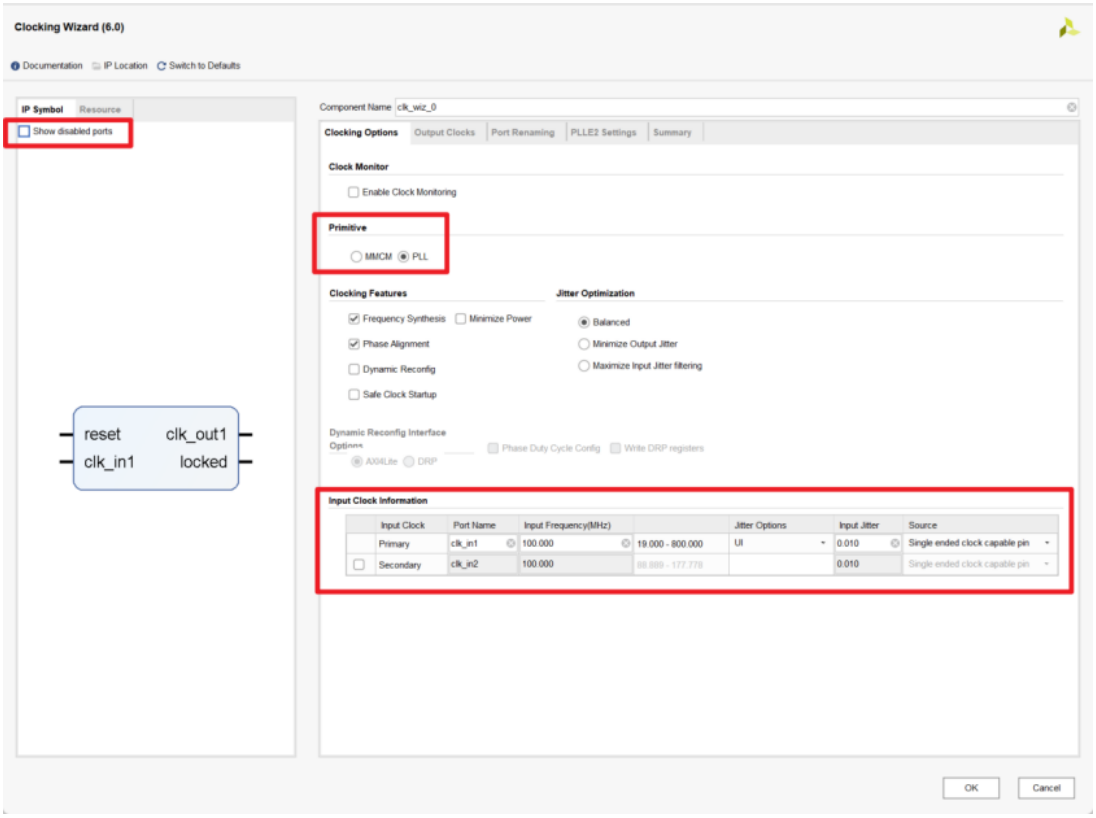


图 E.6: 设置 IP 核

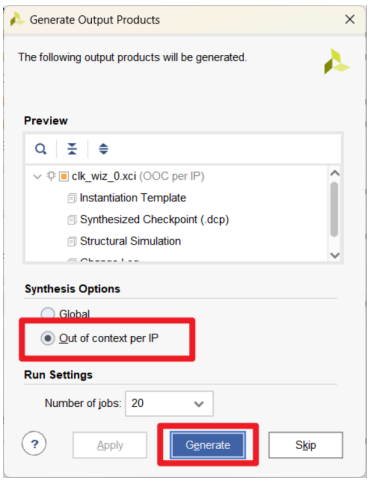


图 E.7: 确认

在例化完成后，可以将 IP 核视为一个普通模块进行使用。其名称即为例化时的 Component Name，在本例子中，即为 `clk_wiz_0`；其输入输出端口可见例化时的 IP Symbol 部分，左侧为输入，右侧为输出。

F | VGA 显示示例（仅供参考

VGA（Video Graphics Array）视频图形阵列是 IBM 于 1987 年提出的一个使用模拟信号的电脑显示标准。VGA 接口即电脑采用 VGA 标准输出数据的专用接口。VGA 接口共有 15 针，分成 3 排，每排 5 个孔，显卡上应用最为广泛的接口类型，绝大多数显卡都带有此种接口。它传输红、绿、蓝模拟信号以及同步信号 (水平和垂直信号)。

使用 Verilog HDL 语言对 VGA 进行控制一般只需控制行扫描信号、列扫描信号和红绿蓝三色信号输出即可。

VGA 输出可分为四个模块：时钟分频模块、数据组织模块、接口控制模块和顶层模块。以下进行分块描述。

时钟模块分频模块对 FPGA 系统时钟进行分频。由于使用的显示屏参数为 640*480*60Hz，其真实屏幕大小为 800*525，因此所需时钟频率为 800*525*60Hz=25.175MHz，可近似处理为 25MHz。FPGA 系统时钟为 100M，因此将其四分频即可基本满足显示要求。

数据组织模块是将预备输出的数据组织为可以通过 VGA 接口控制的数据形式，本次设计中因接口已经协调，数据可不经过此模块进行组织，故可忽略该模块。

接口控制模块通过 VGA 接口对显示屏进行控制。VGA 的扫描顺序是从左到右，从上到下。例如在 640X480 的显示模式下，从显示器的左上角开始往右扫描，直到 640 个像素扫完，再回到最左边，开始第二行的扫描，如此往复，到第 480 行扫完时即完成一帧图像的显示。这时又回到左上角，开始下一帧图像的扫描。如果每秒能完成 60 帧，则称屏幕刷新频率为 60Hz。宏观上，一帧屏幕由 480 个行和 640 个列填充而成，而实际上，一帧屏幕除了显示区，还包含其他未显示部分，作为边框或者用来同步。具体而言，一个完整的行同步信号包含了左边框、显示区、右边框还有返回区四个部分，总共 800 个像素，其分配如下：

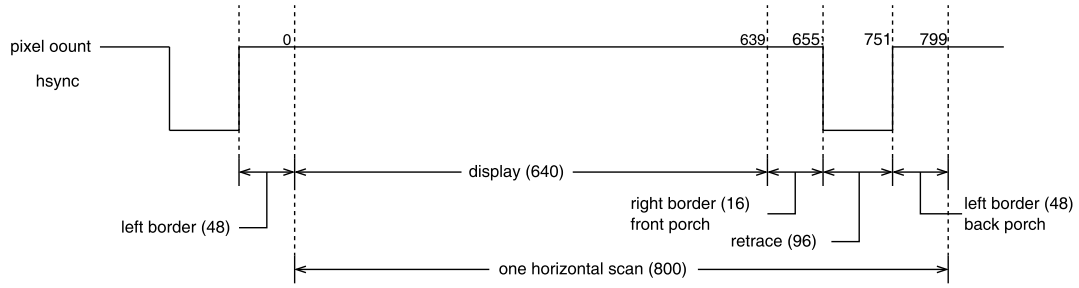


图 F.1: VGA 行扫描时序

同样的，一个完整的垂直同步信号也分为四个区域，总共 525 个像素，分配如下：

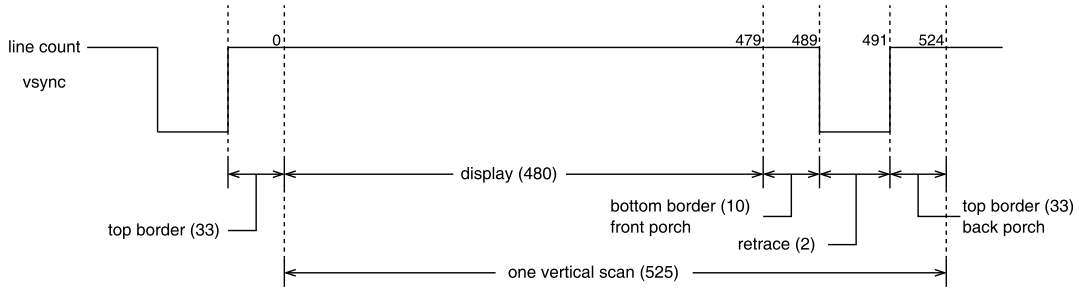


图 F.2: VGA 场扫描时序

模块通过组织输出行扫描信号、列扫描信号和三原色信号对显示屏实现控制。

示例代码:

```
module VGA(
  clk_n,
  rst,
  hsync_r,
  vsync_r,
  OutRed,
  OutGreen,
  OutBlue,
  num
  );
    input clk_n;
    input rst;
    input [199:0] num;
    output reg hsync_r;
        output reg vsync_r;
    output[3:0] OutRed;
    output[3:0] OutGreen;
    output[3:0] OutBlue;
wire [9:0] R [19:0];
assign R[0] = num[9:0];
assign R[1] = num[19:10];
assign R[2] = num[29:20];
assign R[3] = num[39:30];
assign R[4] = num[49:40];
assign R[5] = num[59:50];
……
endmodule
```

VGA 时序控制:

```
reg[9:0]xsync,ysync;
  always @(posedge clk_n or posedge rst) begin
    if (rst) begin
      xsync <= 10'd0;
    end
    else if (xsync == 10'd799) begin
      xsync <= 10'd0;
    end
    else begin
      xsync <= xsync + 1;
    end
  end

  always @(posedge clk_n or posedge rst) begin
    if (rst) begin
      ysync <= 10'd0;
    end
    else if (ysync == 10'd524) begin
```

```
      ysync <= 10'd0;
    end
    else if (xsync == 10'd799) begin
      ysync <= ysync + 1;
    end
  end

……

  wire valid;
  assign valid = (xsync > 143) && (xsync < 784) && (ysync > 34) && (ysync < 515);

  // 将VGA显示屏按X，Y轴方向划分成20*10个方块
  wire [9:0]x_pos, y_pos;
  assign x_pos = xsync - 143;
  assign y_pos = ysync -34;

wire [9:0] x;
wire [19:0] y;

assign x[0] = (x_pos >= 201) && (x_pos <= 224);
assign x[1] = (x_pos >= 225) && (x_pos <= 248);
assign x[2] = (x_pos >= 249) && (x_pos <= 272);
assign x[3] = (x_pos >= 273) && (x_pos <= 296);
……

assign y[0]  = (y_pos >= 1) && (y_pos <= 24);
assign y[1]  = (y_pos >= 25) && (y_pos <= 48);
assign y[2]  = (y_pos >= 49) && (y_pos <= 72);
assign y[3]  = (y_pos >= 73) && (y_pos <= 96);
assign y[4]  = (y_pos >= 97) && (y_pos <=120);
assign y[5]  = (y_pos >= 121) && (y_pos <=144);
……


parameter high = 12'b1111_1111_1111;
  reg [11:0] vga_rgb;
  always @(posedge clk_n or posedge rst) begin
    if (rst) begin
      vga_rgb <= 0;
    end
    else if (valid)
    begin
        if (x_pos>=201 && x_pos<=440)
            if (x[0]&y[0]&R[0][0])
                vga_rgb <= high;
            else if (x[1] & y[0] &R[0][1])
                vga_rgb <= high;
            else if (x[2] & y[0] &R[0][2])
                vga_rgb <= high;
```

```
            else if (x[3] & y[0] &R[0][3])
                 vga_rgb <= high;
            ……
            else
                vga_rgb <= 12'b0000_0000_1111;
         else
            vga_rgb <= 12'b0000_0000_0000;
    end
    else
    begin
      vga_rgb <= 0;
    end
end
// 然后输出RGB值。
  assign OutRed = vga_rgb[11:8];
  assign OutGreen = vga_rgb[7:4];
  assign OutBlue = vga_rgb[3:0];
```