Digital Logic 2022S Project (Tangram) Report

Monad

2022 年 6 月 2 日

目录

1 开发计划及执行记录

1.1 进度安排及执行情况

- **第 11 周**

  参考一些开源项目的项目结构，开始设计并建立项目主体。

- **第 12 周**

  添加 480P VGA 的时钟和时序模块，并做了简单仿真。

  添加 480P VGA 从 RAM 读取数据并显示的逻辑，并通过模拟 RAM 模块进行测试，取得了预期结果。

  设计将每个图形的相关参数储存在 RAM 中，并编写了相关的读、写模块。

  编写输入消抖模块。

  添加了一个基于行扫描的渲染模块。

  添加了一个简单的正方形控制模块（demo）。

- **第 13 周**

  摸了一整个星期。

- **第 14 周**

  从上周实验课在机房的 Vivado 测试，发现 480P 整幅图像的存储所需要的 RAM 大小超过了开发板的 RAM 大小，遂放弃用 RAM 缓存图像的设计。再者由另一个小组的同学反应说 480P 不能在机房的显示器上显示，故将计划的分辨率调整至 600P。

  将原有的 480P 时序切换至 600P。

  使用定点数的方式表示小数，并用泰勒公式编写了 `sin` 三角函数模块。

  添加了其它数学函数，如 `radians` 将角度转成弧度，`sin_deg` 用于求解角度下的三角函数，`cos_deg` 调用 `sin` 模块进行 cos 运算。

  增加了一个 2×2 的矩阵乘法模块。

  用三角函数模块和矩阵乘法模块组合出了一个旋转模块。

  编写了一个不包含除法的 `div10` 模块，将一个数除以 10，同时取余数。

  基于上述旋转模块，编写了一个渲染模块（用于判断某个点是否需要被渲染）。

  在 14 周的实验课上，成功地在机房的显示器上显示了竖条纹（测试图案）。

- **第 15 周**

  添加用于调试的数码管显示模块。

  添加了核心控制模块 `core`，有处理输入移动等主要逻辑。

  增加了一些图形（如等边三角形、平行四边形）。

  增加精确控制坐标功能。

  通过预处理 sin 和矩阵乘法等初始参数，提高渲染时的运算速度，并占用更少的电路资源。

  修复了一些 bug。

  在 15 周的实验课上，成功地跑了起来，完成了答辩。

2 设计

2.1 需求分析

2.1.1 系统功能

- VGA 显示
- 显示基本图形（并且填充是 bonus）
- 图形的旋转和平移
- 调色板（自己添加的功能）

2.1.2 输入设备

开发板上的拨码开关和按钮。

2.1.3 输出设备

VGA 图像输出，和数码管显示。

2.1.4 端口规格

每个拨码开关和按钮都是输入一个 1 比特的 01 信号。其中 rst 按钮较为特殊，其按下时为低电平，平时为高电平。

VGA 输出两个 1 比特的同步信号，以及三个 4 比特的信号分别代表 RGB 通道。

2.2 系统设计

当时首先看到需要绘制基本图形，并且把图形填充有 bonus。想一想，发现用直线绘制图形跟图形填充没有什么相关性，不如直接冲图形填充，写直线绘制还会多一点工作量。然后又看到了旋转，旋转的话无非是弱一点的每次旋转 90 度和更顺滑一点的 360 度旋转。考虑到前者的效果不怎么好看，而且需要分类讨论 4 种方向，在乘上有不同的图形，也不是很有动力去写。于是就直接去搞 360 度旋转了。

这个 Project 主要就是 VGA 渲染图形。VGA 渲染图形，本质上就是要确定某一个像素点应该是什么颜色。然后有这么两种思路：

- 用 RAM 缓存图形渲染：先把图形输出到 RAM，然后再将 RAM 的缓存输出到 VGA。好处是渲染图形的时候可以用任意顺序输出，比较自由。
- 在 VGA 输出的时候，实时决定某个点的颜色。这种方法的好处就是不需要处理 RAM，但是运算部分可能会比较麻烦（基本意味着只能用组合电路或者不超过 2 周期的处理）。

由于后者的缺点，我们一开始用的是前者，即使用 RAM 缓存图像。但是在实际上准备上板的时候，发现储存 600P 画面所需要的 RAM 为 5.49 MBit，大大超出了板子的 RAM 大小 2MBit。所以这个方法就被炮决掉了，只能用后者了。

所以的话我们就需要实时计算某个像素是否需要被绘制，即某个像素是否在某个图形内部。这个计算的话，我们一开始准备手推数学公式，然后推到一半发现可以用线性代数里面的旋转矩阵，确实方便不少。

这样的话，我们就需要实现以 sin 为代表的三角函数，并且需要支持小数运算，然后也需要写一个矩阵乘法。再者就是还有 VGA 显示，就完成了。

然后为了方便调试，还可以顺便弄一个数码管显示，显示图形的坐标等信息。

具体的运行逻辑是基于数据驱动的。我们主要在 core 模块中存储数据，其它模块都是用来处理或显示数据，并不保存重要数据，这样的话整个项目的主次和数据流就会变得十分清楚。

在 core 模块中，我们储存能表达一个图形的参数：坐标 x, y，类型 ty，大小 size，角度 angle 和颜色 color。然后其它逻辑都是围绕读取这些数据并显示，和根据输入更新数据。

还有一些其它的数据，比如调色板中，当前光标的位置 cx, cy；图形的总数 number。

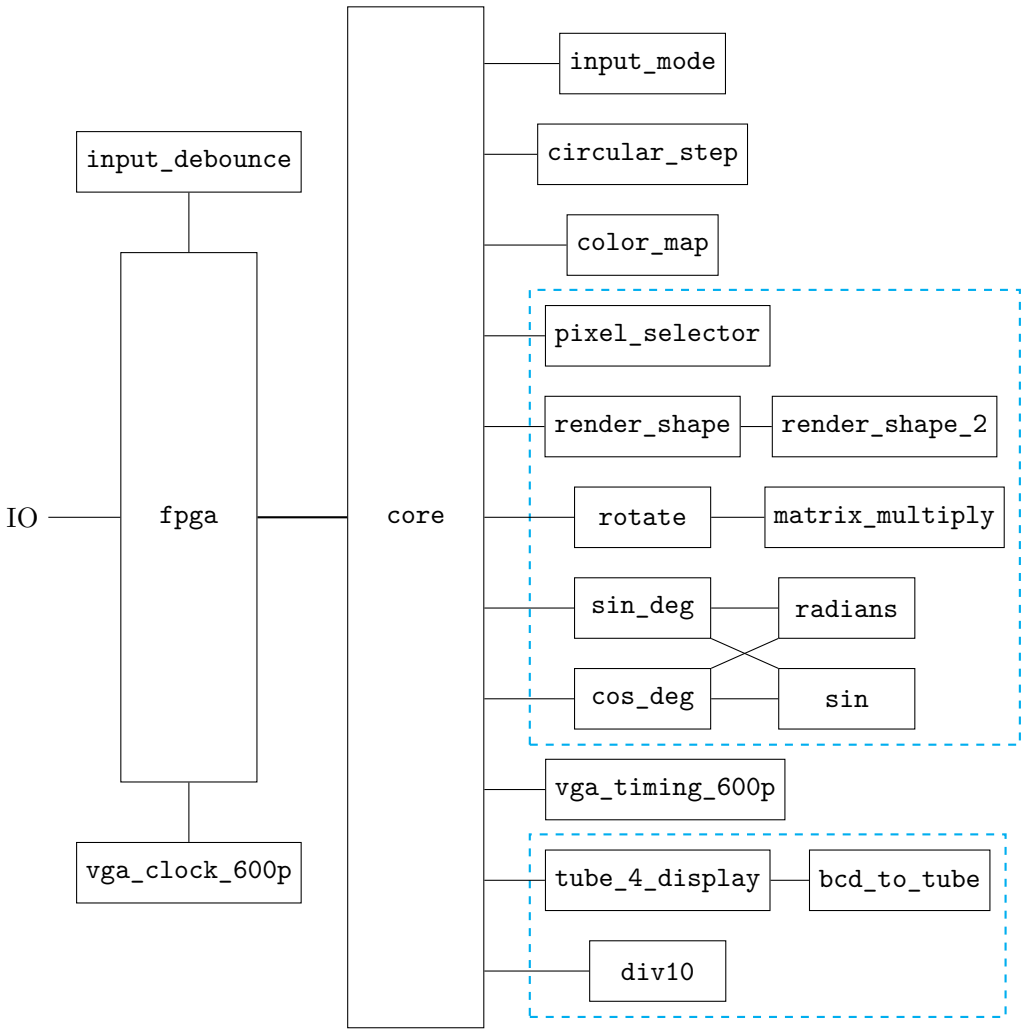
有一个例外，对于 VGA 时序的相关数据，例如虚拟坐标 sx, sy 等数据，由于需要在时序的模块中进行大量更新、外部只需要读取，将它们储存在子模块中是更加合适的选择。

所以总的逻辑就是，当一帧画面渲染完成之后，就“唤醒”主逻辑，开始根据外部输入开始更新数据（比如响应移动图形、变换大小等操作）。更新之后，就可以渲染下一帧。渲染一帧的时候，就是把当前坐标 sx, sy 传入到相关的图形渲染模块中，让它们来决定当前这个点是否需要绘制，或者是需要绘制什么颜色。

事实上，后来为了优化性能，我们把传入坐标给子模块，修改成传一个时钟信号给子模块（一时钟周期代表渲染一个像素），让它们每个时钟周期进行更新。相应地，为了给它们一个初始值，我们就需要在 core 模块中进行一些初始化操作。有关优化的细节，可以看下面的模块分析的 shape_render 部分。

2.3 系统结构

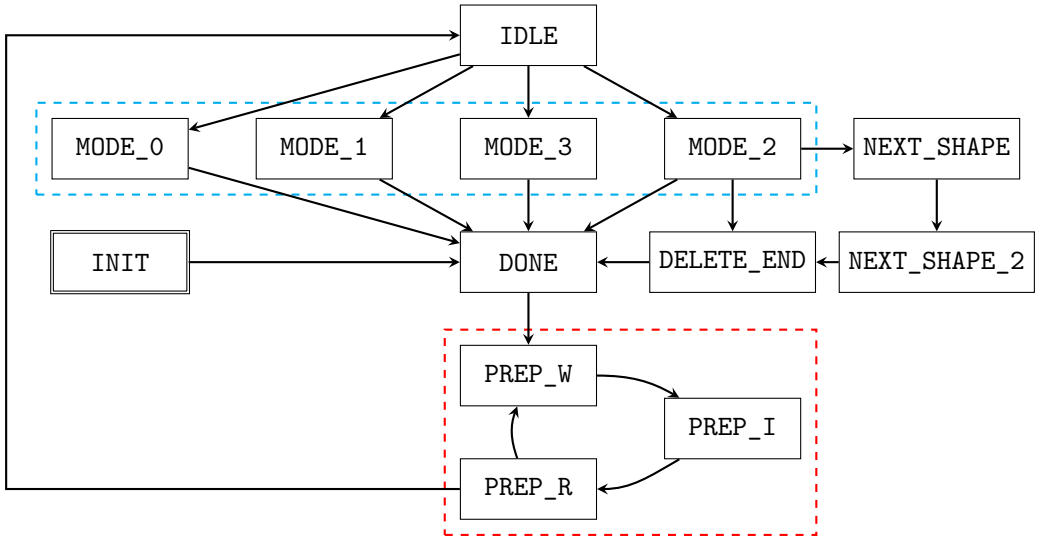
2.3.1 模块结构图



上图中的 `fpga` 模块主要是用于处理与硬件相关的东西，比如输入消抖 `input_debounce`、生成 VGA 时钟 `vga_clock_600p`。并且把 IO 的线连接到核心模块 `core`。

然后核心模块 `core` 包含主控制逻辑，具体逻辑可见下一部分。第一个虚线框里面的模块，负责协同渲染图形。第二个虚线框的模块，负责数码管的显示。不同模块的具体功能，可转至下面的模块分析。

2.3.2 核心控制模块状态图



该状态图对应的是处理输入的流程。处理输入的部分与帧对齐，即每一帧处理一次输入，并更新图像位置等参数，再根据这些更新之后的参数渲染图像。

核心状态大部分时间都处于 IDLE 状态，表示不进行输入按钮的处理，只是在渲染图像。

然后当一帧渲染完成之后，由 vga_timing_600p 发出信号，core 根据拨码开关的开关状态进入不同的模式。其中，

- MODE_0 移动模式：上下左右 4 个按钮分别控制图形的上下左右移动。
- MODE_1 修改模式：上下按钮控制图形的放大与缩小，左右按钮控制图形的旋转，中间的按钮可以切换不同的图形类型（如三角形、正方形等）。
- MODE_2 切换模式：中间按钮切换选择不同的图形，左右两个按钮分别为删除和增加图形。
- MODE_3 调色模式：显示调色板，用上下左右 4 个按钮控制指针选择颜色，按中间按钮确认。

然后 MODE_2 需要更多时间周期进行一些额外的更新操作，故设计了 NEXT_SHAPE 等几个额外的状态。

以上更新完成之后，流程就会来到 DONE。在 DONE 中，核心模块会通知其它模块更新，比如 input_mode 会重置 once 输出。

DONE 完成之后，由于一些优化（在下面的 render_shape 会提及），把一些参数的初始化提前到了核心模块中，需要给每个图形初始化 sin, cos, ix, iy 等参数。其中 PREP_W 是把输入写入给其它模块，在 PREP_W 把数据读出，储存到寄存器中。然后 PREP_I 的存在，是因为 sin 等模块需要大量的乘法操作，从输入到输出所需要的时间会超过 1 个时钟周期，故需要添加一个中间状态来等待。

然后初始化完成之后，就回到了 IDLE，表示完成输入处理，可以渲染。

至于 INIT，由于在初始状态进来的时候，我们也需要初始化一些参数，所以这里把 INIT 指向 DONE。

2.4 模块分析

- **开始之前：关于小数**

  本项目的某些部分需要使用小数（比如 sin 相关的计算），由于浮点数的表示与处理过于复杂，所以这里决定采用定点数的方式来表示。

具体来说，就是对于一个小数，我们可以用一个 8 位的二进制来储存它（实际位宽可以根据需要调整），其中高 4 位用来储存整数部分，低 4 位储存小数部分。更具体地，我们用 $x \times 2^{-Y}$ 来表示一个小数，其中 Y 是一个确定的常量，x 为实际储存的值。上述例子中，$Y = 4$，x 的位宽是 8，能够储存范围约 $[-8, 8]$ 的小数，精度为 $2^{-4} = 0.0625$。

举个例子，对于小数 3.125，其乘上 2^4 之后就是 50，即二进制 `0011_0010`，我们储存这个二进制就行了。

然后对于加法，由于 $x_1 \times 2^{-Y} + x_2 \times 2^{-Y} = (x_1 + x_2) \times 2^{-Y}$，直接加就可以了，减法也是。

对于乘法，$(x_1 \times 2^{-Y}) \cdot (x_2 \times 2^{-Y}) = (x_1 \cdot x_2) \times 2^{-2Y} = (x_1 \cdot x_2 \times 2^{-Y}) \times 2^{-Y}$，就是把 x_1 和 x_2 相乘之后，还要右移 Y 位，才是最终的结果。

- **开始之前：关于常量**

  为了提高的可维护性和可读性，代码使用宏定义了一些常量，包括但不限于

  - `FLOAT_BITS = 32`：小数的总比特数
  - `FLOAT_INT_BITS = 16`：小数的整数部分的比特数
  - `FLOAT_DCM_BITS = 16`：小数的小数部分的比特数
  - `FLOAT_TEMP_BITS = 48`：进行乘法时所需的临时位宽
  - `INT_BITS = 16`：整数的位数
  - `PI`：π 的值
  - `HALF_PI`：$\frac{\pi}{2}$ 的值
  - `INV_6`：$\frac{1}{6}$ 的值
  - `DEGREE_TO_RADIAN`：$\frac{\pi}{180}$ 的值
  - ...

  为了方便和可维护，这些宏定义的具体数值，均使用代码来自动生成。具体细节可以参考 `constants.py` 文件。

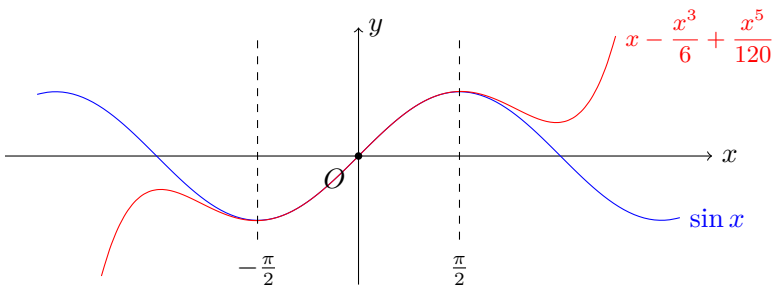
  后文若无具体说明，输入输出的位宽，都是根据该变量的类型（整数或浮点数），在上述宏中选择相应的位宽。所以在源代码中，可以根据某个输入输出的位宽定义，了解其储存的数值类型。

- **`math/sin` 三角函数模块**

  输入：范围在 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ 的弧度 θ。**输出**：$\sin\theta$ 的值。

  由于硬件语言过于底层，没有可以直接使用的 sin 函数，所以一个比较方便的方法就是通过泰勒公式来计算 sin 函数的值。

  通过观察泰勒公式与真实函数的图像，不难发现，在 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ 范围中，泰勒公式取前三项就误差不大了。



所以这里就取泰勒公式的前三项

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120}$$

并且可以微调 120 处的具体数值，平衡函数两边的误差，使得在指定输入范围内，相对误差不超过 0.07%。

具体代码如下所示。其中 `theta3` 表示 θ 的 3 次方，以此类推。然后 `INV_6` 为 $\frac{1}{6}$ 的值（常量），可以避免昂贵的除法运算。

```
module sin (
    input  wire logic signed [`FLOAT_BITS-1:0] in,
    output      logic signed [`FLOAT_BITS-1:0] out
);
    logic signed [`FLOAT_TEMP_BITS-1:0] theta2;
    logic signed [`FLOAT_TEMP_BITS-1:0] theta3;
    logic signed [`FLOAT_TEMP_BITS-1:0] theta5;
    logic signed [`FLOAT_TEMP_BITS-1:0] part2;
    logic signed [`FLOAT_TEMP_BITS-1:0] part3;

    always_comb begin
        theta2 = (in * in) >>> (`FLOAT_DCM_BITS);
        theta3 = (theta2 * in) >>> (`FLOAT_DCM_BITS);
        theta5 = (theta2 * theta3) >>> (`FLOAT_DCM_BITS);

        part2 = (theta3 * (`INV_6)) >>> (`FLOAT_DCM_BITS);
        part3 = (theta5 * (`INV_120)) >>> (`FLOAT_DCM_BITS);
        out = in - part2 + part3;
    end
endmodule
```

- **`math/radians` 角度转弧度模块**

  输入：一个整数角度。**输出**：该角度对应的弧度。

  非常简单，直接将角度乘上 $\frac{\pi}{180}$ 即可。

```
module radians (
    input  wire logic signed [  `INT_BITS-1:0] in,
    output      logic signed [`FLOAT_BITS-1:0] out
);
    assign out = in * (`DEGREE_TO_RADIAN);
endmodule
```

- **`math/sin_deg` 角度下的三角函数模块**

  输入：范围在 $[-180, 180]$ 的角度。**输出**：该角度下的 sin 函数值。

  由于 `sin` 模块只接受 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ 范围的输入，所以这里还需要根据输入的角度，经过三角变换，将角度变换至 $[-90, 90]$ 度内。（而且事实表明，在角度下做三角变换，比在弧度下进行三角变换，所带来的误差更小。）

  具体的三角变换如下：

$$\sin\theta = \sin\phi, \quad \text{where } \phi = \begin{cases} -\theta + 180^\circ & ,\theta > 90^\circ \\ -\theta - 180^\circ & ,\theta < -90^\circ \\ \theta & ,\text{other situation} \end{cases}$$

在代码中，即

```
module sin_deg (
    input  wire logic signed [  `INT_BITS-1:0] in,
    output      logic signed [`FLOAT_BITS-1:0] out
);
    logic signed [  `INT_BITS-1:0] in2;
    logic signed [`FLOAT_BITS-1:0] theta;

    radians radians(.in(in2), .out(theta));
    sin sin(.in(theta), .out);

    always_comb
        if (in > 16'sd90)
            in2 = 16'sd180 - in;
        else if (in < -16'sd90)
            in2 = - (16'sd180) - in;
        else
            in2 = in;

endmodule
```

- **math/cos_deg 角度下的三角函数模块**

  输入：范围在 $[-180, 180]$ 的角度。**输出**：该角度下的 cos 函数值。

  与 sin_deg 相似，先通过三角函数变换至与 sin 相关的角度，再调用 sin 进行计算。即

$$\cos\theta = \sin\phi, \quad \text{where } \phi = \begin{cases} 90^\circ + \theta & ,\theta < 0 \\ 90^\circ - \theta & ,\theta > 0 \end{cases}$$

```
module cos_deg (
    input  wire logic signed [  `INT_BITS-1:0] in,
    output      logic signed [`FLOAT_BITS-1:0] out
);
    logic signed [`INT_BITS-1:0] in2;
    logic signed [`FLOAT_BITS-1:0] theta;

    radians radians(.in(in2), .out(theta));
    sin sin(.in(theta), .out);

    always_comb
        if (in < 0)
            in2 = 16'sd90 + in;
        else
            in2 = 16'sd90 - in;
endmodule
```

- **math/matrix_multiply 矩阵乘法模块**

  输入：若干个小数输入。**输出**：输出矩阵乘法的结果。（具体是哪些输入输出，可参考下面的公式。）

$$\begin{bmatrix} \texttt{u1} & \texttt{u2} \end{bmatrix} \begin{bmatrix} \texttt{a11} & \texttt{a12} \\ \texttt{a21} & \texttt{a22} \end{bmatrix} = \begin{bmatrix} \texttt{v1} & \texttt{v2} \end{bmatrix}$$

用矩阵乘法法则直接暴算即可。

```
module matrix_multiply (
    input  wire logic signed [`FLOAT_BITS-1:0] u1,
    input  wire logic signed [`FLOAT_BITS-1:0] u2,
    input  wire logic signed [`FLOAT_BITS-1:0] a11,
    input  wire logic signed [`FLOAT_BITS-1:0] a12,
    input  wire logic signed [`FLOAT_BITS-1:0] a21,
    input  wire logic signed [`FLOAT_BITS-1:0] a22,
    output      logic signed [`FLOAT_BITS-1:0] v1,
    output      logic signed [`FLOAT_BITS-1:0] v2
);
    logic signed [`FLOAT_TEMP_BITS-1:0] b1, b2, b3, b4;

    always_comb begin
        b1 = (u1 * a11) >>> (`FLOAT_DCM_BITS);
        b2 = (u2 * a21) >>> (`FLOAT_DCM_BITS);
        v1 = b1 + b2;

        b3 = (u1 * a12) >>> (`FLOAT_DCM_BITS);
        b4 = (u2 * a22) >>> (`FLOAT_DCM_BITS);
        v2 = b3 + b4;
    end
endmodule
```

- `render/rotate` **旋转模块**

  输入：`x`, `y` 表示原坐标，`sin`, `cos` 表示 $\sin\theta$ 和 $\cos\theta$。

  输出：变换后的坐标 `x1`, `y1`。

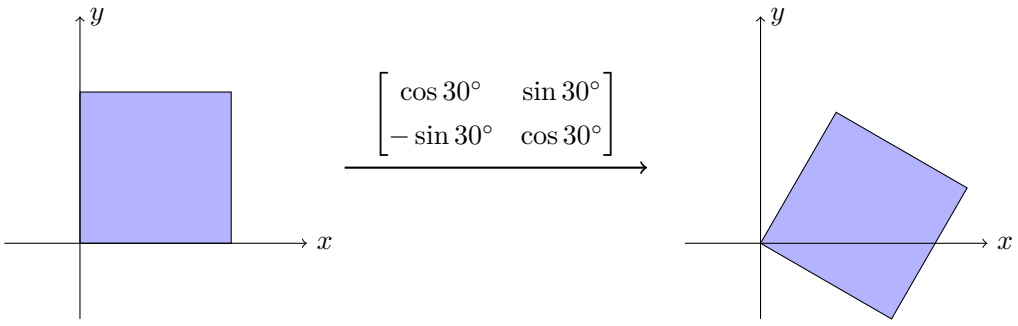
  原来该模块是输入 `x`, `y` 和旋转的角度 `theta`（整数）。后来为了性能优化，把三角函数的运算提取到了外部，所以就变成了 `sin` 和 `cos`。

  该模块的功能，是把坐标 (x, y)，以 $(0, 0)$ 为中心，顺时针旋转一定的角度（实际上后面调用的时候是属于逆变换，所以实际显示效果是逆时针），输出变换后的坐标 (x_1, y_1)。

  具体的实现，由线性代数的旋转变换，可知

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \end{bmatrix}$$

  即



于是调用矩阵乘法，把参数填好就行了。（虽然这个不是标准的变换矩阵的格式，但是随便吧，不想改了）

```
module rotate (
    input  wire logic signed [`FLOAT_BITS-1:0] x,
    input  wire logic signed [`FLOAT_BITS-1:0] y,
    input  wire logic signed [`FLOAT_BITS-1:0] sin,
    input  wire logic signed [`FLOAT_BITS-1:0] cos,
    output      logic signed [`FLOAT_BITS-1:0] x1,
    output      logic signed [`FLOAT_BITS-1:0] y1
);
    matrix_multiply m(
        .u1(x), .u2(y),
        .a11(cos), .a12(sin), .a21(-sin), .a22(cos),
        .v1(x1), .v2(y1)
    );
endmodule
```

- **render/render_shape_basic 基础图形渲染模块**

  输入：x, y 表示当前渲染的坐标，ty, size 表示图形的类型和大小。

  输出：out，表示当前坐标是否需要被渲染。

  这个模块是一个基础的渲染模块，在一个非常理想情况下，判断某个点是否需要被渲染。这个“理想的情况”具体指图形的参考坐标位于 $(0,0)$，无旋转。

  具体实现就根据不同的图形，先手动算出它的几个边界，然后把不等式写进程序里即可。

  例如正方形的一个点是否需要被渲染，等价于 $x \geq 0$ && $y \geq 0$ && $x <$ size && $y <$ size。

  具体代码如下所示。

```
module render_shape_basic (
    input  wire logic signed [`INT_BITS-1:0] x,
    input  wire logic signed [`INT_BITS-1:0] y,
    input  wire logic        [`INT_BITS-1:0] ty,
    input  wire logic        [`INT_BITS-1:0] size,
    output      logic out
);
    logic signed [`INT_BITS-1:0] sum, sub, hy;

    always_comb begin
        sum = x + y;
        sub = x - y;
        hy = y >>> 1;

        case (ty)
            0: out = (x >= 0) && (y >= 0) && (sum < size);  // 三角形
            1: out = (y >= 0) && (hy <= x) && (x + hy < size);  // 正三角形
            2: out = (x >= 0) && (y >= 0) && (x < size) && (y < size);  // 正方形
            3: out = (x >= 0) && (y >= 0) && (x < size) && (hy < size);   // 1:2 矩形
            4: out = (y >= 0) && (y < size) && (sum >= 0) && (sum < size);
            5: out = (y >= 0) && (y < size) && (sub >= 0) && (sub < (size << 1));
            default: out = 0;
        endcase
    end
endmodule
```

- **render/render_shape 更完善的图形渲染模块**

  在开始这个模块之前，我们先来看看之前的代码。

```
module render_shape (
    input  wire logic        [`INT_BITS-1:0] ty,
    input  wire logic signed [`INT_BITS-1:0] x0,
    input  wire logic signed [`INT_BITS-1:0] y0,
    input  wire logic        [`INT_BITS-1:0] size,
    input  wire logic signed [`INT_BITS-1:0] angle,

    input  wire logic signed [`INT_BITS-1:0] x,
    input  wire logic signed [`INT_BITS-1:0] y,

    output      logic out
);
    logic signed [`INT_BITS-1:0] dx, dy, sum;
    logic signed [`FLOAT_BITS-1:0] fx, fy, rx, ry;
    logic signed [`FLOAT_BITS-1:0] cos, sin;

    cos_deg cos(.in(angle), .out(cos));
    sin_deg sin(.in(angle), .out(sin));

    rotate rotate(
        .x(fx), .y(fy),
        .sin, .cos,
        .x1(rx), .y1(ry)
    );

    render_shape_basic basic(
        .x(rx[`FLOAT_BITS-1:`FLOAT_DCM_BITS]),  // 选择整数部分
        .y(ry[`FLOAT_BITS-1:`FLOAT_DCM_BITS]),
        .ty,
        .size,
        .out
    );

    always_comb begin
        dx = x - x0;  // (1)
        dy = y - y0;
        fx = { dx, { `FLOAT_DCM_BITS{1'b0} } };
        fy = { dy, { `FLOAT_DCM_BITS{1'b0} } };
    end
endmodule
```

其输入一个图形的相关参数（包括参考点和旋转角度），然后判断某个点是否需要被渲染。

我们可以考虑将这些复杂的情况做一些调整，使其符合 `render_shape_basic` 的理想情况，然后调用即可。

首先，我们需要把图形的参考点变换至 $(0,0)$，此处做一下平移即可（代码为 `(1)` 处）。

然后，变换至 $(0,0)$ 之后，我们就可以调用 `rotate` 做旋转变换了。

变换完成之后，我们就得到了理想坐标 $(\texttt{rx},\texttt{ry})$，然后调用 `render_shape_basic` 判断即可。

但是，这样有个问题，就是每个图形都会使用一个矩阵乘法模块，和两个三角函数模块。这样就会导致在渲染大量图形的时候，就需要非常多的乘法模块，在开发板上的 DSP 模块被使用完成之后，就会使用 LUT 进行乘法运算，很快就会超支。实测用这种方式，只能支持同时渲染最多 3 个图形。

观察代码实现，不难看出，`sin` 和 `cos` 的值，在整一帧的渲染中都不会发生变化，是一个定值。这样我们就可以把它挪到外面，用一对三角函数提前计算所有图形所需要的 `sin` 和 `cos` 值，之后只用传入 `sin` 和 `cos` 的值给渲染模块即可。

对于矩阵乘法，也不难发现，渲染的时候每次都是 x 每次递增 1，扫完一行之后，x 归零，然后 y 增加 1。利用这个规律性的变化，可以把矩阵乘法的乘法，优化成每次坐标变换都给结果加上一个 sin 或 cos，变乘为加。不过这种优化，也需要在外面预处理：当渲染 $(0,0)$ 坐标的时候，变换之后的坐标是在哪里（代码中的 (1) 处）。

于是我们就有了最终的模块。

```
module render_shape (
    input  wire logic clk,       // 时钟信号，每周期渲染一个像素
    input  wire logic newline,   // 新的一行，使 x 归零，y 加一
    input  wire logic newframe,  // 新的一帧，使 x y 归零

    input  wire logic        [`INT_BITS-1:0] ty,
    input  wire logic        [`INT_BITS-1:0] size,
    input  wire logic signed [`FLOAT_BITS-1:0] sin,
    input  wire logic signed [`FLOAT_BITS-1:0] cos,
    input  wire logic signed [`FLOAT_BITS-1:0] ix,  // (1)
    input  wire logic signed [`FLOAT_BITS-1:0] iy,

    output      logic out
);
    logic signed [`INT_BITS-1:0] sum, sub, hy;
    logic signed [`FLOAT_BITS-1:0] x, y, rx, ry;

    always_ff @(posedge clk) begin
        if (newframe) begin
            x <= ix;
            y <= iy;
            rx <= ix - sin;
            ry <= iy + cos;
        end else if (newline) begin
            x <= rx;
            y <= ry;
            rx <= rx - sin;
            ry <= ry + cos;
        end else begin
            x <= x + cos;
            y <= y + sin;
        end
    end

    render_shape_basic basic(
        .x(x[`FLOAT_BITS-1:`FLOAT_DCM_BITS]),
        .y(y[`FLOAT_BITS-1:`FLOAT_DCM_BITS]),
        .ty,
        .size,
        .out
    );
endmodule
```

- `render/color_map` 渲染调色板

  输入：x, y 表示当前选择的颜色位于调色板的 (x,y) 处，sx, sy 表示当前屏幕渲染位置。

  输出：color 表示选择的颜色，render 表示该处渲染的颜色。

  这个模块主要是用于渲染一个调色板，每 2×2 个像素表示一个颜色，然后用坐标组合起来表示不同颜色。同时在当前选择的位置渲染一个白色的十字，显示出当前位置。

```
module color_map #(
    parameter PIXLW = 12  // 3 * 4 = 12 位颜色
) (
    input  wire logic [`INT_BITS-1:0] x,
    input  wire logic [`INT_BITS-1:0] y,
    input  wire logic [`INT_BITS-1:0] sx,
    input  wire logic [`INT_BITS-1:0] sy,
    output      logic [PIXLW-1:0] color,
    output      logic [PIXLW-1:0] render
);
    logic signed [`INT_BITS-1:0] dx, dy;
    logic [PIXLW-1:0] t_render;

    assign t_render = { sy[6:1], sx[6:1] };
    assign color = { y[6:1], x[6:1] };

    always_comb begin
        dx = sx - x;
        dy = sy - y;

        if (x == sx && y == sy)  // 中心位置十字留空
            render = t_render;
        else if (x == sx && (-3 <= dy && dy <= 3))
            render = 12'hFFF;
        else if (y == sy && (-3 <= dx && dx <= 3))
            render = 12'hFFF;
        else
            render = t_render;
    end
endmodule
```

- `math/div10` **除以 10 模块**

  输入：一个被除数。**输出**：`quotient` 为商，`remainder` 为余数。

  使用 Hacker's Delight 实现，用位运算等基本运算完成除以 10 并取余数的效果。

```
module div10 (
    input  wire logic [`INT_BITS-1:0] in,
    output      logic [`INT_BITS-1:0] quotient,
    output      logic [3:0] remainder
);
    logic [`INT_BITS-1:0] q, r;

    always_comb begin
        q = (in >> 1) + (in >> 2);
        q = q + (q >> 4);
        q = q + (q >> 8);
        q = q + (q >> 16);
        q = q >> 3;
        r = in - (((q << 2) + q) << 1);
        quotient = q + (r > 9);
        remainder = in - (quotient << 1) - (quotient << 3);
    end
endmodule
```

- `tube/bcd_to_tube` **BCD 转 7 段数码管**

  输入：一个 BCD8421 数字。**输出**：对应的数码管的信号（如果超出范围，则输出全 0）。

  打表即可。

值得一提的是，bcd_to_tube 的输入，位宽是 4，正好可以与 div10 的余数直接连起来。

```
module bcd_to_tube (
    input  wire logic [3:0] in,
    output      logic [6:0] out   // GFEDCBA
);
    always_comb begin
        case (in)
            0: out = 7'b0111111;
            1: out = 7'b0000110;
            2: out = 7'b1011011;
            3: out = 7'b1001111;
            4: out = 7'b1100110;
            5: out = 7'b1101101;
            6: out = 7'b1111101;
            7: out = 7'b0000111;
            8: out = 7'b1111111;
            9: out = 7'b1101111;
            default: out = 7'b0;
        endcase
    end
endmodule
```

- **tube/tube_4_display 数码管显示不同数字的模块**

  输入：clk 为时钟信号，4 个 in 表示各个数字的显示信号，dp 表示 4 个小数点的信号。

  输出：en 表示目前激活哪个位置，out 表示这个激活的位置的信号（含 P）。

  通过快速切换不同的数码管，使 4 个数字快速地依次显示，达到显示 4 个不同数字的目的。

  目前是每隔 2^{16} 个时钟周期切换一个数字，在 40MHz 下就是 1.64 ms 切换一个数字，经试验，效果良好。

```
module tube_4_display (
    input  wire logic clk,

    input  wire logic [3:0] in1,
    input  wire logic [3:0] in2,
    input  wire logic [3:0] in3,
    input  wire logic [3:0] in4,
    input  wire logic [3:0] dp,

    output      logic [3:0] en,
    output      logic [7:0] out   // PGFEDCBA
);
    logic [15:0] cnt;
    logic [3:0] d;
    logic [1:0] id;

    bcd_to_tube to(.in(d), .out(out[6:0]));

    always_ff @(posedge clk) begin
        if (&cnt)
            id <= id + 1;
        cnt <= cnt + 1;
    end

    assign out[7] = dp[id];
    always_comb begin
        case (id)
            2'd0: d = in1;
            2'd1: d = in2;
            2'd2: d = in3;
            2'd3: d = in4;
        endcase

        en = 4'b0001 << id;
    end
endmodule
```

- **input/input_debounce 输入消抖模块**

  此模块的逻辑参考了 ProjectF: FPGA Pong 相关代码的实现。

  其主要思路是用一个 cnt 计数器来记录按钮按下的时间。如果当前按钮状态与输入信号不一致，则将 cnt 加 1，否则将其置为 0。当 cnt 计满后，将自身的按钮状态翻转。

  如果令 cnt 为 18 位的寄存器，则在 40MHz 的时钟信号下，其达到最大值所需要的时间约为 6.55ms，即外部信号稳定 6.55ms 之后，这个模块才会更新输出信号。

  输入：clk 为时钟信号，in 为输入信号。

  输出：out 表示消抖之后的信号，ondn 在输出信号转为“按下”的时候发出一个持续一个时钟周期的信号，onup 同理。

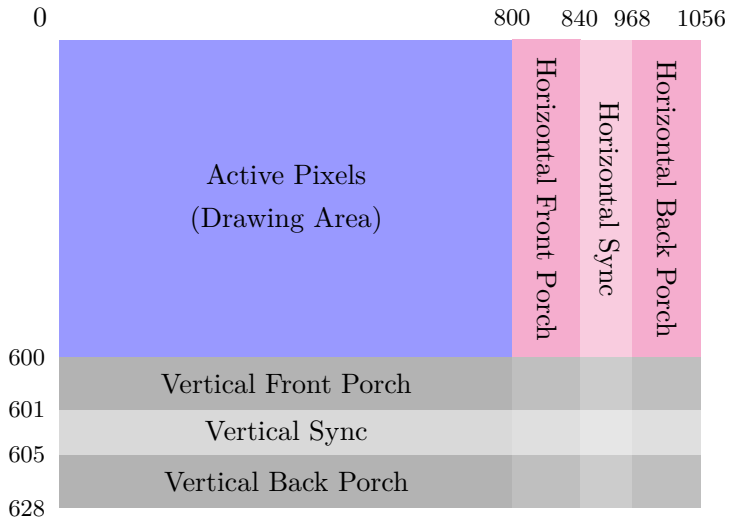
```
module input_debounce (
    input  wire logic clk,      // clock
    input  wire logic in,       // signal input
    output      logic out = 0,  // signal output (debounced)
    output      logic ondn,     // on down (one tick)
    output      logic onup      // on up (one tick)
);
    // sync with clock and combat metastability
    logic sync_0, sync_1;
    always_ff @(posedge clk) sync_0 <= in;
    always_ff @(posedge clk) sync_1 <= sync_0;

    logic [17:0] cnt = 0;
    logic idle, max;
    always_comb begin
        idle = (out == sync_1);
        max  = &cnt;
        ondn = (~idle) & max & (~out);
        onup = (~idle) & max & out;
    end

    always_ff @(posedge clk) begin
        if (idle) begin
            cnt <= 0;
        end else begin
            cnt <= cnt + 1;
            if (max)
                out <= ~out;
        end
    end
endmodule
```

- **vga/vga_timing_600p 数码管显示不同数字的模块**

  VGA 的时序构成主要由下图所示。我们可以用虚拟坐标 sx 和 sy 在下面的时序图中扫描。



  如果虚拟坐标位于蓝色区域，那么这个虚拟坐标也等价于正在绘制的位置的坐标，并且把模块的 de 输出信号设为 1，表示现在数据有效。如果虚拟坐标位于 Sync 区域，则将相应的 Sync 信号设为 0（平时为 1）。

  输入：clk_pix 为 40MHz 的时钟信号，rst_pix 为 rst 信号。

输出：sx, sy 表示坐标，hsync, vsync 表示 VGA 的同步信号，de 表示数据是否有效，还输出三个信号供控制模块使用。

```
module vga_timing_600p (
    input  wire logic clk_pix,       // pixel clock
    input  wire logic rst_pix,       // reset in pixel clock domain
    output      logic endframe,      // signal: frame finish
    output      logic newframe,      // signal: new frame
    output      logic newline,       // signal: new line
    output      logic [9:0] sx,      // horizontal screen position
    output      logic [9:0] sy = 0,  // vertical screen position
    output      logic hsync,         // horizontal sync
    output      logic vsync,         // vertical sync
    output      logic de             // data enable
);
    // horizontal timings
    parameter HA_END = 800;           // end of active pixels
    parameter HS_STA = HA_END + 40;   // sync starts after front porch
    parameter HS_END = HS_STA + 128;  // sync ends
    parameter LINE   = 1056-1;        // last pixel on line

    // vertical timings
    parameter VA_END = 600;           // end of active pixels
    parameter VS_STA = VA_END + 1;    // sync starts after front porch
    parameter VS_END = VS_STA + 4;    // sync ends
    parameter SCREEN = 628-1;         // last line on screen

    logic [10:0] ix = 0;

    always_comb begin
        sx = ix[9:0];
        hsync = ~((ix >= HS_STA) && (ix < HS_END));
        vsync = ~((sy >= VS_STA) && (sy < VS_END));
        de = (ix < HA_END) && (sy < VA_END);

        endframe = (ix == HA_END) && (sy == VA_END);
        newframe = (ix == 0) && (sy == 0);
        newline = (ix == 0);
    end

    always_ff @(posedge clk_pix) begin
        if (rst_pix) begin
            ix <= 0;
            sy <= 0;
        end else if (ix == LINE) begin
            ix <= 0;
            sy <= (sy == SCREEN) ? 0 : sy + 1;
        end else begin
            ix <= ix + 1;
        end
    end
endmodule
```

- **control/pixel_selector**

  由于会有多个图形同时渲染，所以我们需要根据一组 shape_render 输出的一组 out 信号，来确定当前像素是属于哪个图形（或者哪个都不属于）。

  输入：en 为上文所说的不同图形 out 信号组成的，位宽与最大图形数量相等。

  输出：black 表示当前像素不属于任何一个图形，id 表示所属的图形的编号（与 en 对应），multiple 表示是否有多个图形在此处重叠。

`multiple` 可以用位运算判断一个数是否为 2 的幂的方法来确定。

```
module pixel_selector #(
    parameter MAXSHP = 16
) (
    input  wire logic [MAXSHP-1:0] en,
    output      logic black,
    output      logic [`INT_BITS-1:0] id,
    output      logic multiple
);
    assign multiple = |(en & (en - 1));
    assign black = ~(|en);

    always_comb begin
        casex (en)
            16'b0000000000000001: id = `INT_BITS'd0;
            16'b000000000000001x: id = `INT_BITS'd1;
            16'b00000000000001xx: id = `INT_BITS'd2;
            16'b0000000000001xxx: id = `INT_BITS'd3;
            16'b000000000001xxxx: id = `INT_BITS'd4;
            16'b00000000001xxxxx: id = `INT_BITS'd5;
            16'b0000000001xxxxxx: id = `INT_BITS'd6;
            16'b000000001xxxxxxx: id = `INT_BITS'd7;
            16'b00000001xxxxxxxx: id = `INT_BITS'd8;
            16'b0000001xxxxxxxxx: id = `INT_BITS'd9;
            16'b000001xxxxxxxxxx: id = `INT_BITS'd10;
            16'b00001xxxxxxxxxxx: id = `INT_BITS'd11;
            16'b0001xxxxxxxxxxxx: id = `INT_BITS'd12;
            16'b001xxxxxxxxxxxxx: id = `INT_BITS'd13;
            16'b01xxxxxxxxxxxxxx: id = `INT_BITS'd14;
            16'b1xxxxxxxxxxxxxxx: id = `INT_BITS'd15;
            default: id = `INT_BITS'bx;
        endcase
    end
endmodule
```

- `control/input_mode` **按钮模式控制**

  我们设计了一个功能，可以由某个拨码开关控制按钮的模式：按下一次只移动一格，或按下多久就移动多远。

  输入：`clk` 时钟信号，`in`, `down` 分别是 `input_debounce` 的 `out` 和 `ondn` 信号，`mode` 表示按钮模式，`clr` 表示重设 `down` 信号。

  输出：`out` 表示最终输出信号，`once` 表示上述模式的前者（有的功能需要忽略模式切换，只使用该模式）。

```
module input_mode (
    input  wire logic clk,
    input  wire logic in,
    input  wire logic down,
    input  wire logic mode,
    input  wire logic clr,
    output      logic out,
    output      logic once = 0
);
    assign out = mode ? once : in;

    always_ff @(posedge clk) begin
        if (clr) begin
            once <= 0;
        end else begin
            if (down)
                once <= 1;
        end
    end
endmodule
```

- `control/circular_step` 循环序列

  有些参数需要循环，比如角度到了 359 度之后，下一度应该回到 0 度。

  输入：`in` 表示当前数值。**输出**：`next`, `prev` 分别表示后一个和前一个值。

  把输入输出定义好之后，剩下的就只剩下两个 if 的事情了。

```
module circular_step #(
    parameter DATAW = `INT_BITS,
    parameter DW_BOUND = -180,
    parameter UP_BOUND = 179
) (
    input  wire logic signed [DATAW-1:0] in,
    output      logic signed [DATAW-1:0] prev,
    output      logic signed [DATAW-1:0] next
);
    assign next = (in == UP_BOUND) ? DW_BOUND : (in + 1);
    assign prev = (in == DW_BOUND) ? UP_BOUND : (in - 1);
endmodule
```

- `control/core` 核心模块

  核心模块主要是处理输入，并且更新图形的参数，初始化图形渲染。

  然后状态转移就像核心控制模块状态图那样子。

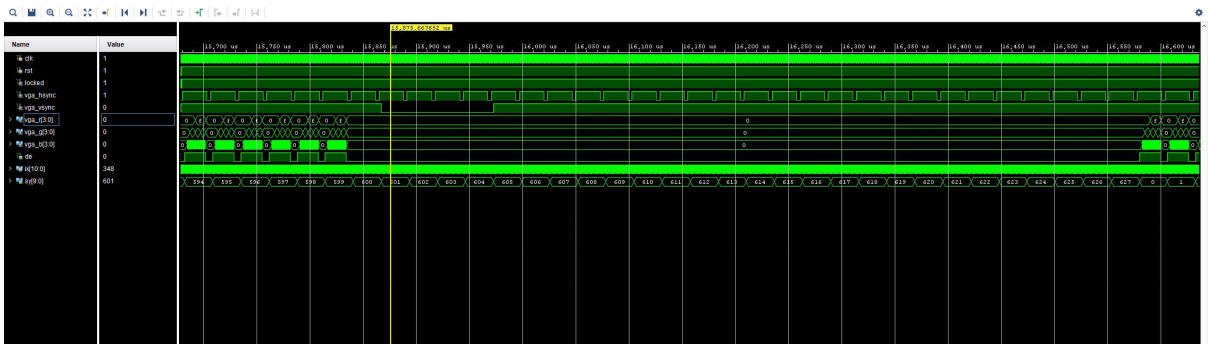
```
enum {
    INIT, IDLE,
    PREP_W, PREP_I, PREP_R,
    MODE_0, MODE_1, MODE_2, MODE_3,
    NEXT_SHAPE, NEXT_SHAPE_2, DELETE_END,
    DONE
} state, next_state;

if (rst) begin
    next_state = INIT;
end else begin
    case (state)
        INIT: next_state = DONE;
        IDLE: begin
            if (sg_endframe) begin
                if (swch_3)      next_state = MODE_3;
                else if (swch_1) next_state = MODE_1;
                else if (swch_2) next_state = MODE_2;
                else             next_state = MODE_0;
            end else
                next_state = IDLE;
        end
        PREP_W: next_state = PREP_I;
        PREP_I: next_state = (&a_cnt) ? PREP_R : PREP_I;
        PREP_R: next_state = (a_id == MAXSHP - 1) ? IDLE : PREP_W;
        MODE_0: next_state = DONE;
        MODE_1: next_state = DONE;
        MODE_2: begin
            if (l_btn_once)      next_state = DELETE_END;
            else if (c_btn_once) next_state = NEXT_SHAPE;
            else                 next_state = DONE;
        end
        MODE_3: next_state = DONE;
        NEXT_SHAPE: next_state = NEXT_SHAPE_2;
        NEXT_SHAPE_2: next_state = DELETE_END;
        DELETE_END: next_state = DONE;
        DONE: next_state = PREP_W;
        default: next_state = DONE;
    endcase
end
```

然后至于每个状态的具体逻辑，由于代码太长了，就不贴在这里了，可以去源代码翻翻。然后 `core` 模块还有大量的子模块实例化，还有一大堆的连线，所以可能看起来会比较长。

3 仿真测试

3.1 VGA 仿真



如上图所示是 VGA 仿真的结果，这里选取了较为标志性的一段，即场同步信号工作的一段。

上图中场同步信号（即 `vga_vsync`）在 `sy` 从 601 到 604 为低电平，正好对应了 VGA 时序中的 Vertical Sync。

	Horizontal	Vertical
Active Pixels	800	600
Front Porch	40	1
Sync	128	4
Back Porch	88	23
Total	1056	628

而且 Vertical Sync 与 Horizontal Sync 的相对位置与 ProjectF 所给出的例子相似。

3.2 `cocotb` 单元测试

我们觉得看仿真波形图来调试的方式局限性太大了，尤其是对于有小数输入输出的数学模块来说，手动将整数翻译回小数是一件非常麻烦的事情。而且，仿真成功只能说明此时此刻的模块是正确的，对于一些模块，后面进行进一步修改的时候，除非再手动看一遍波形图，否则就是裸奔；但是一旦模块多起来，不断地看波形图就逐渐变成了累赘，效率非常底下。

为了解决这个问题，我们使用一个用 Python 编写的 `cocotb` 库对模块进行单元测试。它可以在 Python 中设置给模块的输入，然后等待一定时钟周期后，再读取其输出，然后在 Python 下检验其正确性。

这样就有一个好处，我们可以使用脚本生成很多测试用例，然后自动化地给模块进行魔鬼测试。以 `sin_deg` 为例，我们可以很方便地测试从 $[-180, 180]$ 范围内的 360 个角度，并且由脚本自动化的生成和解析定点小数，十分方便。

而且对模块进行修改之后，可以直接由一条命令运行单元测试，快捷而稳定地测试代码的正确性。

以 `div10` 为例：

```python
@cocotb.test()
async def div10_test(dut):
    dut_in = getattr(dut, 'in')

    for i in range(0, 10000):
        dut_in.value = i
        await Timer(2, units='ns')

        quotient = dut.quotient.value.integer
        remainder = dut.remainder.value.integer

        assert quotient == i // 10, f'quotient is wrong: {quotient}'
        assert remainder == i % 10, f'remainder is wrong: {remainder}'
```

整段代码除去空行就只有 10 行代码，不需要任何注释，可读性就已经非常高了。它对于 $[0, 10000)$ 中的每一个整数，都将其传给 `div10` 模块，然后等待 2ns 让模块进行计算后，再读取其商 `quotient` 和余数 `remainder`，最后用 `assert` 断言商和余数就是对应输入的商和余数。

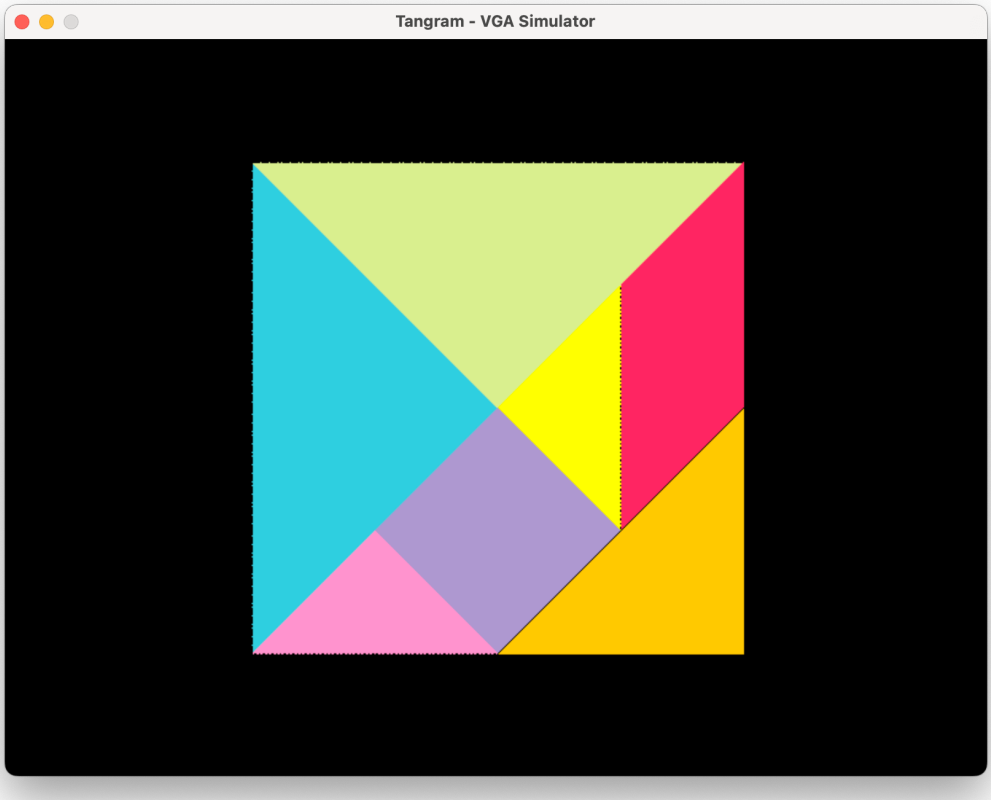
```
$ make test
/Applications/Xcode.app/Contents/Developer/usr/bin/make -f Makefile results.xml
MODULE=test_math_div10 TESTCASE= TOPLEVEL=div10 TOPLEVEL_LANG=verilog \
         /opt/homebrew/bin/vvp -M /Users/monad/Projects/Tangram/venv/lib/python3.8/site-packages/cocotb/
libs -m libcocotbvpi_icarus   sim_build/sim.vvp
     -.--ns INFO     cocotb.gpi                         ..mbed/gpi_embed.cpp:110  in set_program_name_in_venv
Using Python virtual environment interpreter at /Users/monad/Projects/Tangram/venv/bin/python
     -.--ns INFO     cocotb.gpi                         ../gpi/GpiCommon.cpp:99   in gpi_print_registered_impl
VPI registered
     0.00ns INFO     Running on Icarus Verilog version 11.0 (stable)
     0.00ns INFO     Running tests with cocotb v1.6.2 from /Users/monad/Projects/Tangram/venv/lib/python3.8/
site-packages/cocotb
     0.00ns INFO     Seeding Python random module with 1654014041
     0.00ns INFO     Found test test_math_div10.div10_test
     0.00ns INFO     running div10_test (1/1)
 20000.00ns INFO     div10_test passed
 20000.00ns INFO     ****************************************************************************************
                     ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                     ****************************************************************************************
                     ** test_math_div10.div10_test     PASS       20000.00           0.26      75481.11  **
                     ****************************************************************************************
                     ** TESTS=1 PASS=1 FAIL=0 SKIP=0              20000.00           0.27      74420.01  **
                     ****************************************************************************************
```

这一个测试，只需要 `make test` 一个命令，就可以在高达 1 秒的时间内，检验对于 10000 个不同的输入，模块的正确性。

其与波形图孰优孰劣不言而喻。

3.3 VGA 模拟器

Hack 了一下底层一些参数，用 `pygame` 弄出了一个 VGA 模拟器。不过速度极慢，基本上两三分钟才能渲染一帧。不过用来测试第一帧初始画面也足够了。



4 总结与心得体会

4.1 Project 完成情况

感觉还行，至少 Project 题目上所给出的要求（包括 bonus）大部分都实现了（除了碰撞停止这个确实有点复杂之外）。而且还弄了一些自己觉得好玩的功能。

4.2 心得体会

其实 VGA 这个项目的困难，主要是体现在设计上面，就是题目只是给了你所需要实现的效果，并没有给出中间具体应该如何设计。所以由于条条大路通罗马，有很多方式可以实现这个项目。但是如果还要考虑到用哪条路比较省力，就需要在前期进行大量的思考。然后这个我觉得在没有一定的项目经验的情况下是比较难判断的。

同时，由于 mac 上不能跑 Vivado，而且 Vivado 的项目文件夹也不适合版本管理，所以在前期也先花了点时间在 GitHub 上找一些开源项目的项目结构，并且也找了一些测试工具和方法。所以我们决定，在 `rtl` 文件夹中存储不同的模块文件，然后在 `tests` 里面放一些测试文件。测试是基于 Python 的 `cocotb` 库以及仿真工具 iVerilog，可以用 Python 来写测试，比肉眼看波形图友善不少。而且在写 `sin` 等数学模块的时候，涉及到定点小数更加不可能用肉眼看。

上面这个就是前期的准备工作。

其实我们没有着急开始写项目，我们从 12 周才开始正式开始写代码，而且 13 周还摸了一周。盲目内卷是没有意义的，与其开卷，不如先停下来，看看走哪条路，用什么设计，怎么也比走错路舒服。

而且实际上，前期花时间确定的项目结构和测试方法，在后面添加数学函数的测试的时候，给了不少帮助。后来把整体设计搬到数学函数处理旋转的时候，也是先完成设计，代码层面都是行云流水，没有多大阻力。

然后现在回去看整体的代码，发现全部的 verilog 代码加起来还不到 1000 行（然后有一半是 `core` 模块），代码量其实不算高。

```
$ cloc .
      65 text files.
      65 unique files.
      21 files ignored.

-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Verilog-SystemVerilog           22            207             28            957
Python                          20            253             50            564
make                            16             48              0            166
C/C++ Header                     1              6              1             20
-------------------------------------------------------------------------------
SUM:                            59            514             79           1707
-------------------------------------------------------------------------------
```

整个项目写下来，虽然挑战非常多，包括用矩阵乘法处理旋转的方法，都是不是一开始就想好的，但是把这些挑战一个一个地解决掉，最终呈现出来一个十分炫酷的显示结果的时候，还是非常有成就感的。