南京大学

数电实验九：

字符输入界面

课程名称：　数字逻辑与计算机组成实验

姓名：　孙文博

学号：　201830210

班级：　数电一班

邮箱：　201830210@smail.nju.edu.cn

实验时间：　2022.5.20 - 2022.5.25

一、 实验目的

1. 了解字模点阵相关的知识；
2. 综合运用存储器、键盘、显示器的相关知识，实现多个模块之间的交互和接口设计；
3. 掌握大型（相对而言）工程文件的 debug 方法；

二、 实验环境

设计\编译环境：Quartus (Quartus Prime 17.1) Lite Edition

开发平台：DE10-Standard

FPGA 芯片：Cyclone II 5CSXFC6D6

VGA 显示器、PS\2 键盘

三、 实验原理

1. 字符显示方法

字符显示界面只在屏幕上显示 ASCII 字符，其所需的资源比较少。首先，ASCII 字符用 7bit 表示，共 128 个字符。大部分情况下，我们会用 8bit 来表示单个字符，所以一般系统会预留 256 个字符。我们可以在系统中预先存储这 256 个字符的字模点阵，如下图所示：

图 9-1: ASCII 字符字模

这里每个字符高为 16 个点，宽为 9 个点。因此单个字符可以用 16 个 9bit 数来表示，每个 9bit 数代表字符的一行，对应的点为“1”时显示白色，为“0”时显示黑色。因此，我们只需要 256 × 16 × 9 ≈ 37kbit 的空间即可存储整个点阵。

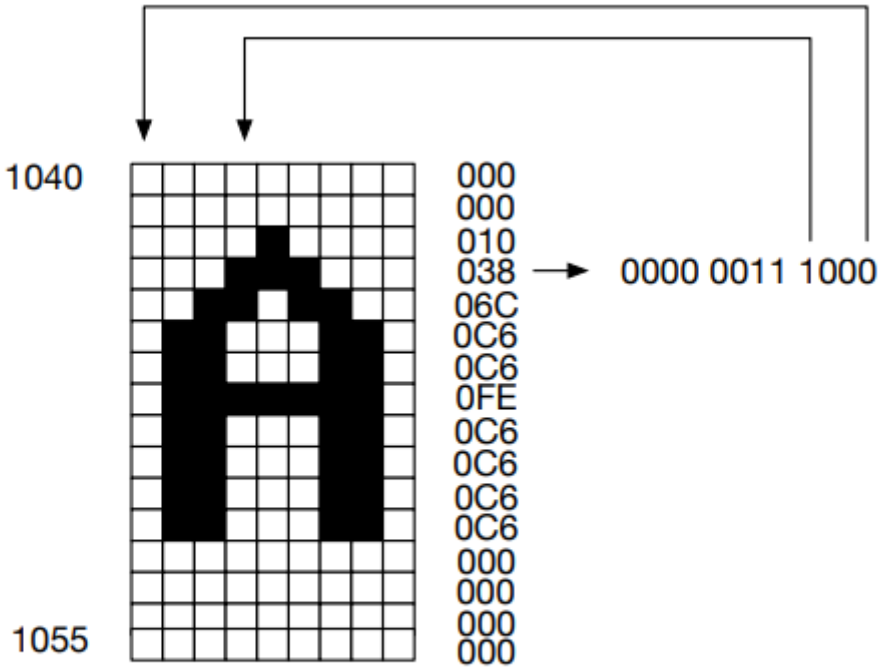


图 9-2: 字模“A”与存储器的关系

有了字符点阵后，系统就不再需要记录屏幕上每个点的颜色信息了，只需要记录屏幕上显示的 ASCII 字符即可。在显示时，根据当前屏幕位置，确定应该显示那个字符，再查找对应的字符点阵即可完成显示。对于 640×480 的屏幕，可以显示 30 行（30×16=480），70 列（70×9=630）的 ASCII 字符。系统的显存只需要 30 × 70 大小，每单元存储 8bit 的 ASCII 字符即可。

2. 扫描显示方法

我们之前已经实现了 VGA 控制模块，该模块可以输出当前扫描到的行和列的位置信息，我们只需要稍加改动，即可让其输出当前扫描的位置对应 30×70 字符阵列的坐标（0 ⩽ x ⩽ 69，0 ⩽ y ⩽ 29）。利用该坐标，我们可以查询字符显存，获取对应字符的 ASCII 编码。利用 ASCII 编码，我们可以查询对应的点阵 ROM，再根据扫描线的行和列信息，可以知道当前扫描到的是字符内的哪个点。这时，可以根据该点对应的 bit 是 1 还是 0，选择输出白色还是黑色。

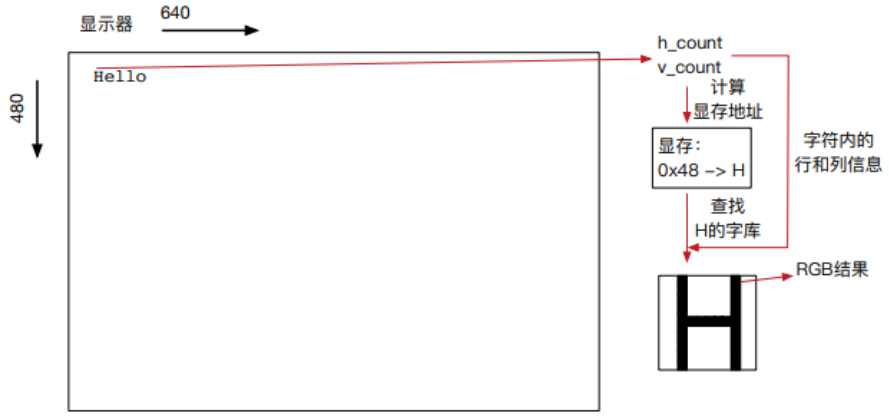


图 9-3: 字符显示流程示意图

我们将显示的过程总结如下：

1.根据当前扫描位置，获取对应的字符的 x,y 坐标，以及扫描到单个字符点阵内的行列信息；

2.根据字符的 x,y 坐标，查询字符显存获取对应 ASCII 编码；

3.根据 ASCII 编码和字符内的行信息，查询点阵 ROM 获取对应行的 9bit 数据；

4.根据字符内的列信息，取出对应的 bit，并根据该 bit 设置颜色。

3. 显存读写方法

对于键盘输入，我们可以复用之前实现的键盘控制器。在键盘有输入的时候对字符显存进行改写，将按键对应的 ASCII 码写入显存的合适位置，这样输入就可以直接反馈到屏幕上了。

四、 实验过程

1. 设计思路

实现字符输入交互界面是一个大工程，我们需要综合运用存储器、键盘控制、显示器输出等多个模块的内容，如果糅合在一起编写，直到写完之后再去测试，这会使 debug 变得非常困难，所以我们必须把这个大工程拆分成几个子工程，实现完一部分就测试一部分，保证先前编写的模块不出问题，再去实现下一部分。

对于本实验而言，拆分出两个子工程最为合适。因为关于键盘输出，我们已经在实验八中大致实现了，只要稍作修改并观察按键输出

是否正确即可；而关于显示器方面，显示器所要输出字符的 ASCII 码都保存在一个 30×70 的存储器里，对它进行修改需要在同一个 always 块中实现，无法拆分出更多模块（子工程）。扫描显示屏幕的模块也可以作为一个子工程，但是该模块已经在实验九的题目中提供了，不需要我们进行调试，所以我们把它合并到了显示器相关的子工程中。因此，我们最终把本实验的工程拆分为两个子工程，一个子工程输出键盘按键的 ASCII 码，以及 CapsLock、Shift、方向键等控制键的按键情况，另一个子工程则根据这些输出改写显示屏幕的存储器，然后反馈到显示器上。

2. 键盘设计过程

首先，我们把键盘分为三种键，定义如下：

有效字符键：按下后屏幕上会增加相应字符的键，如字母、数字、符号等。

无效字符键：按下后屏幕输出以及光标都不发生改变的键，如 CapsLock、Shift、Ctrl、F0~F12 等。

非字符屏幕变化键：除去有效字符键之外，按下后会使屏幕输出或光标变化的键。本实验中只实现了左方向键、右方向键、回车键和退格键。

对于无效字符键，在本实验中只有大小写与其相关，而其他子工程可以通过 ASCII 码来判断当前需要输出的是大写还是小写，所以我们可以在键盘子工程内处理无效字符键的问题，不需要把它作为键盘

子工程的输出。因此，我们只需要把有效字符键（结合此时无效字符键的按键情况）的 ASCII 码和非字符屏幕变化键作为键盘子工程的输出即可。因此输出端有如下几个信号：

- 当前按键的 ASCII 码，输出到七段数码管 HEX1 和 HEX0 上。
- 无效字符键，输出到 LEDR[3:0]。（一个键对应一个信号灯）
- 非字符屏幕变化键的编号，输出到 LEDR[7:5]。（3 个二进制位，表示 编号 0~7）
- 非字符屏幕变化键的按键信号（按任意该类键即触发有效），输出到 LEDR[9]。

键盘模块的实现代码（部分）如下：

```
always @ (posedge clk) begin
   if (clrn == 0) begin
      nonchar_en <= 0;
      nonchar_key <= 0;
      nextdata_n <= 1;
      pressed <= 1;
      key_off <= 0;
      E0_skip <= 0;
      kbd_type <= 4'b0;
      eff_data <= 8'b0;
      cnt_off <= 0;
   end else begin
      if (ready) begin
         if (data == 8'hF0) begin // don't read "F0"
            pressed <= 0; // skip code "F0"
            nonchar_en <= 0;  // disable nonchar keys
            nonchar_key <= 0;
            eff_data <= 8'b0;
         end else begin
            if (pressed == 0) begin
               // the first time read the code that next "F0"
               pressed <= 1;
               key_off <= 1; // wait a while before reading the next code
               if (data == 8'h58) kbd_type[0] <= ~kbd_type[0]; // Caps
               else if (data == 8'h12 || data == 8'h59) kbd_type[1] <= 0; // shift off
               else if (data == 8'h14) kbd_type[2] <= 0; // ctrl off
               else if (data == 8'h11) kbd_type[3] <= 0; // alt off
            end else if (key_off == 0) begin
               if (data == 8'hE0) begin
                  E0_skip <= 1; // skip code "E0"
                  eff_data <= 8'b0;
               end else begin
                  if (E0_skip) begin
                     // the first time read the code that next "E0"
                     E0_skip <= 0;
                     if (data == 8'hF0) begin
                        pressed <= 0;
```

```
    end else begin
        if (data == 8'h6B) begin // left
            nonchar_key <= 1;
            nonchar_en <= 1;
        end else if (data == 8'h72) begin // down
            nonchar_key <= 2;
            nonchar_en <= 1;
        end else if (data == 8'h75) begin // up
            nonchar_key <= 3;
            nonchar_en <= 1;
        end else if (data == 8'h74) begin // right
            nonchar_key <= 4;
            nonchar_en <= 1;
        end else if (data == 8'h14) kbd_type[2] <= 1; // ctrl
        else if (data == 8'h11) kbd_type[3] <= 1; // alt
    end
end else begin
    // normal data
    if (data == 8'h12 || data == 8'h59
        || data == 8'h14 || data == 8'h11
        || data == 8'h58 || data == 8'h5A
        || data == 8'h66 || data == 8'h76
        || data == 8'h05 || data == 8'h06
        || data == 8'h04 || data == 8'h0C
        || data == 8'h03 || data == 8'h0B
        || data == 8'h83 || data == 8'h0A
        || data == 8'h01 || data == 8'h09
        || data == 8'h78 || data == 8'h07)
    begin
        eff_data <= 8'b0;
        if (data == 8'h12 || data == 8'h59) kbd_type[1] <= 1; // shift
        else if (data == 8'h14) kbd_type[2] <= 1; // ctrl
        else if (data == 8'h11) kbd_type[3] <= 1; // alt
        else if (data == 8'h5A) begin // enter
            nonchar_key <= 5;
            nonchar_en <= 1;
        end else if (data == 8'h66) begin// backspace
```

我们实现完键盘子工程后，下载到 FPGA 开发板上进行测试。观察 FPGA 开发板上相应信号的输出值，在确保无误之后，就可开始下一个子工程的模块实现。

3. 显示器设计过程

根据验收需要，我们设计屏幕的显示如下：

第一行输出“Hello, world!”，第二行输出学号姓名，第三行为空行，我们输入的字符所能出现的位置从屏幕上第四行开始。默认初始屏幕的第四行会有命令提示符“mylinux>”， 光标位于命令提示符之后。键盘的输入对光标所在的位置进行填充字符或修改字符。左右方向键移动光标，向前最多能够移动到命令提示符

之后，向后最多能够移动到有输入的最后一行末尾的非空字符后。我们把这个“有输入的最后一行末尾的非空字符后”的位置记为 L。对于回车键，无论光标位于何处，都会针对位置 L 进行换行，换行后在行首填充命令提示符，然后把光标挪至命令提示符之后。对于退格键，只有当光标位于位置 L 时才会删除光标前一个字符，否则不进行任何操作。当光标和位置 L 都处于屏幕最右下角时，输入字符会使得屏幕清空（第一行和第二行不改变），新输入的字符出现在第三行的第一个位置，光标停留在该位置之后。

显存初始化代码如下（在屏幕上显示初始内容）：

```
// "Hello, world! \^o^/"
screen_ram[0][0] = 8'h48;screen_ram[0][1] = 8'h65;
screen_ram[0][2] = 8'h6C;screen_ram[0][3] = 8'h6C;
screen_ram[0][4] = 8'h6F;screen_ram[0][5] = 8'h2C;
screen_ram[0][7] = 8'h77;screen_ram[0][8] = 8'h6F;
screen_ram[0][9] = 8'h72;screen_ram[0][10] = 8'h6C;
screen_ram[0][11] = 8'h64;screen_ram[0][12] = 8'h21;
screen_ram[0][14] = 8'h5C;screen_ram[0][15] = 8'h5E;
screen_ram[0][16] = 8'h6F;screen_ram[0][17] = 8'h5E;
screen_ram[0][18] = 8'h2F;

// "swb 201830210"
screen_ram[1][0] = 8'h73;screen_ram[1][1] = 8'h77;
screen_ram[1][2] = 8'h62;screen_ram[1][3] = 8'h00;
screen_ram[1][4] = 8'h32;screen_ram[1][5] = 8'h30;
screen_ram[1][6] = 8'h31;screen_ram[1][7] = 8'h38;
screen_ram[1][8] = 8'h33;screen_ram[1][9] = 8'h30;
screen_ram[1][10] = 8'h32;screen_ram[1][11] = 8'h31;
screen_ram[1][12] = 8'h30;

// "mylinux>"
cmd_prompt[0] = 8'h6D;cmd_prompt[1] = 8'h79;
cmd_prompt[2] = 8'h6C;cmd_prompt[3] = 8'h69;
cmd_prompt[4] = 8'h6E;cmd_prompt[5] = 8'h75;
cmd_prompt[6] = 8'h78;cmd_prompt[7] = 8'h3E;

for (i = 0; i < cmd_prompt_size; i = i+1)
   screen_ram[3][i] = cmd_prompt[i];
```

显存实现代码主体是一个 always 语块：

```
    // for continuous input
    always @ (posedge clk) begin
       if (clrn == 0) begin
          flush_clk <= 0;
          flush_cnt <= 0;
       end else if (flush_cnt == flush_freq) begin
          flush_cnt <= 0;
          flush_clk <= ~flush_clk;
       end else flush_cnt <= flush_cnt + 1;
    end

    always @ (posedge flush_clk) begin
       if (!nonchar_en && char_ascii == 8'b0) begin // no operations
          if (blink_cnt == blink_freq) begin // cursor blink rate
             blink_cnt <= 0;
             cursor_blink <= ~cursor_blink;
          end else blink_cnt <= blink_cnt + 1;
       end else if (nonchar_en) begin // special command
          cursor_blink <= 1;
          case (nonchar_key)
             1: begin // left
                if (!(line_with_prompt[cursor_y] && cursor_x == cmd_prompt_size)) begin
                   if (cursor_x == 0) begin
                      if (cursor_y != 2) begin
                         cursor_y <= cursor_y - 1;
                         cursor_x <= 69;
                      end
                   end else cursor_x <= cursor_x - 1;
                end
             end

     4: begin // right
        if (cursor_y < ram_pointer_y
        || cursor_y == ram_pointer_y && cursor_x < ram_pointer_x) begin
           if (cursor_x == 69) begin
              cursor_x <= 0;
              cursor_y <= cursor_y + 1;
           end else cursor_x <= cursor_x + 1;
        end
     end
     5: begin // enter
        if (ram_pointer_y == 29) begin
           for (i = 2; i < 30; i = i+1)
              for (j = 0; j < 70; j = j+1)
                 screen_ram[i][j] = 8'b0;
           for (i = 0; i < cmd_prompt_size; i = i+1)
              screen_ram[2][i] <= cmd_prompt[i];
           line_with_prompt <= 30'b0;
           line_with_prompt[2] <= 1;
           cursor_x <= cmd_prompt_size;
           cursor_y <= 2;
           ram_pointer_x <= cmd_prompt_size;
           ram_pointer_y <= 2;
        end else begin
           for (i = 0; i < cmd_prompt_size; i = i+1)
              screen_ram[ram_pointer_y+1][i] <= cmd_prompt[i];
           line_with_prompt[ram_pointer_y+1] <= 1;
           cursor_x <= cmd_prompt_size;
           cursor_y <= ram_pointer_y + 1;
           ram_pointer_x <= cmd_prompt_size;
           ram_pointer_y <= ram_pointer_y + 1;
        end
     end

  6: begin // backspace
     if (cursor_x == ram_pointer_x && cursor_y == ram_pointer_y
     && (cursor_x != 0 || cursor_y != 2)
     && (cursor_x != cmd_prompt_size || !line_with_prompt[cursor_y])
     ) begin
        if (cursor_x == 0) begin
           screen_ram[cursor_y-1][69] <= 8'b0;
           cursor_x <= 69;
           cursor_y <= cursor_y - 1;
           ram_pointer_x <= 69;
           ram_pointer_y <= cursor_y - 1;
        end else begin
           screen_ram[cursor_y][cursor_x-1] <= 8'b0;
           cursor_x <= cursor_x-1;
           ram_pointer_x <= cursor_x-1;
        end
     end
  end
```

```
end else begin // input char
   if (cursor_x == 69) begin
      if (cursor_y == 29) begin
         for (i = 2; i < 30; i = i+1)
            for (j = 0; j < 70; j = j+1)
               screen_ram[i][j] = 8'b0;
         screen_ram[2][0] <= char_ascii;
         line_with_prompt <= 30'b0;
         cursor_x <= 1;
         cursor_y <= 2;
         ram_pointer_x <= 1;
         ram_pointer_y <= 2;
      end else begin
         screen_ram[cursor_y][cursor_x] <= char_ascii;
         cursor_x <= 0;
         cursor_y <= cursor_y + 1;
         if (ram_pointer_y == cursor_y
         && ram_pointer_x == cursor_x) begin
            ram_pointer_x <= 0;
            ram_pointer_y <= cursor_y + 1;
         end
      end
   end else begin
      screen_ram[cursor_y][cursor_x] <= char_ascii;
      cursor_x <= cursor_x + 1;
      if (ram_pointer_y == cursor_y
      && ram_pointer_x == cursor_x) begin
         ram_pointer_x <= cursor_x + 1;
      end
   end
end
//end
```

这样，我们就完成了显示器相关的子工程。把两个子工程合并成一个大工程，下载到 FPGA 开发板上运行，测试成功则本实验顺利完成。

五、 实验结果

1. 思考题

本次实验暂无思考题。

2. 上板验收

直接下载到 FPGA 开发板上运行，用键盘和显示器进行测试，成功实现以下功能：

基本要求

- 支持所有小写英文字母和数字输入，以及不用 Shift 即可输

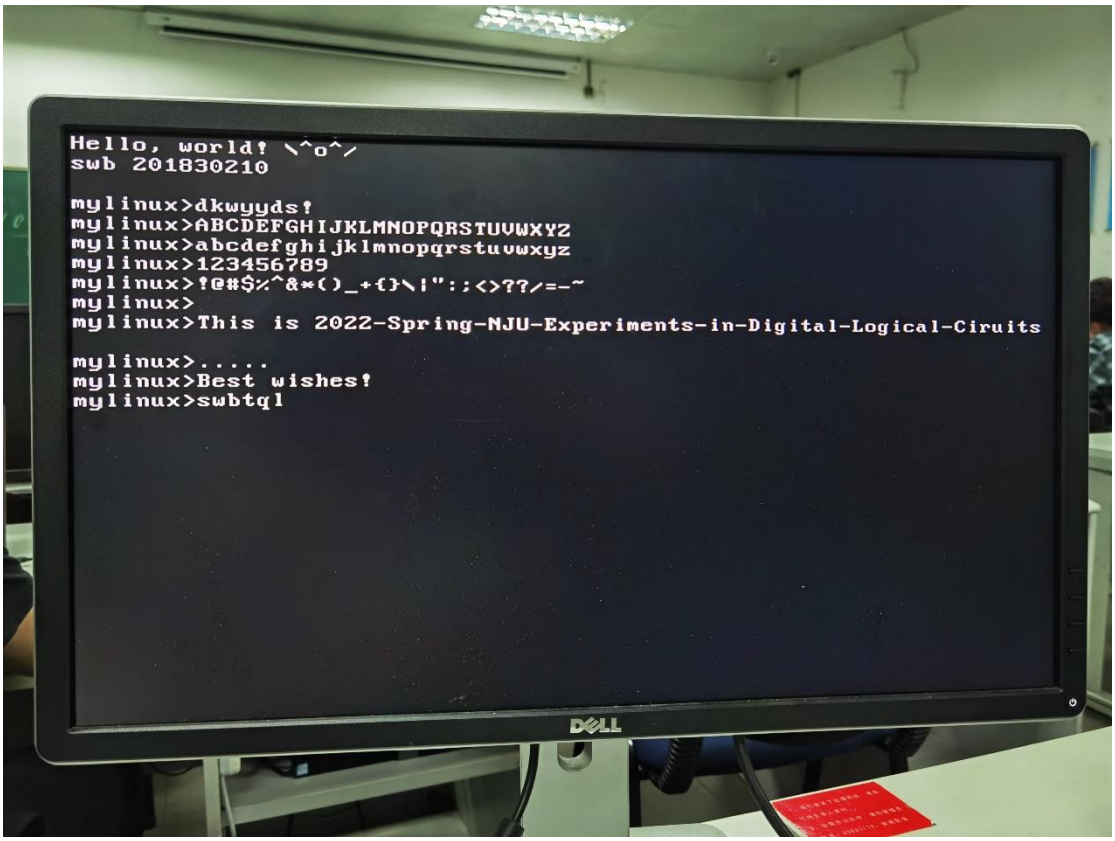
入的符号。

- 一直按压某个键时，重复输出该字符。
- 输入至行尾后自动换行，输入回车也换行。

可选扩展要求

- 可以显示光标，建议可以用显示闪烁的竖线或横线作为光标。
- 支持 BackSpace 键删除光标前的字符。
- BackSpace 删除至本行开始后，再按 BackSpace 可以删除回车键，光标停留在上一行末尾的非空字符后。
- 支持自动滚屏，即输入到最后一行后回车出现新空白行，并且所有已输入的行自动上移一行。
- 支持 Shift 键以及大小写字符输入。
- 支持方向键移动光标。
- 在行首显示命令提示符。

附上验收图片如下：



六、 总结与反思

本次实验综合了前面八九两次实验，完成了键盘的交互和屏幕的显示，最终实现了一个迷你版的虚拟机交互窗口，成就感可以说是直接拉满，但是随之而来的代价是本次实验工程量巨大，且不说合并前两次工程文件需要注意的细节，光是编写顶层文件和交互文件就要花去很多时间，而 debug 环节更是百般折磨。值得一提的是，由于工程量过大，debug 时不得不将每个模块单独分开，也使得整个工程的条理变得比较清晰。这学期的数电实验马上就要迎来结尾了，加油！💪