数字电路与数字系统大实验

MIPS32计算机系统

马英硕 191220080

许希帆 191870219

张宇晨 191220171

2020 秋季学期

前　言

Preface

前言

计算机出现以后，人类对机械计算科学具有了更深刻的认识。在高级语言诞生以前，只有汇编语言和机器语言。经过 11 次 FPGA 编程小实验，我们也许有能力利用电路元件和数字设计思想动手实现一个真正的简单计算机系统（名称为 SSshell）。

本实验报告是对我们小组 3 人实验和实验内容的一个全面、综合的介绍。报告中给出了我们整个设计的所有细节和具体实现，力求在不牺牲完备性和严谨性的前提下，给出比较必要和核心的说明。

报告中第一章给出了有关于本次实验的实验说明，其他章节则是一部分 SSshell 的设计与实现描述。这些描述是采用文字、代码和流程图形式来完成的。

在实验报告中，我们采用了一种循序渐进的方式，希望在叙述后面的部分时可以在前面的部分找到依赖和解释。如果在叙述时必须介绍后面的部分，会在叙述时给出对应章节指代(Reference)。另外，实验时遇到的问题等如果有必要提及，也将展示在报告中。

实验中我们综合了小实验中曾经报告过的内容，并对其进行一些修改。这将重新展示在实验报告中。

本实验报告中，马英硕同学撰写了第 1~4 章，许希帆同学撰写了第 5~6 章，张宇晨同学撰写了第 7~8 章。除了部分原理内容直接展示了部分实验手册中内容，其他内容均为自己撰写。

目　录

Catalog

您可以点击对应标题的内容，来到达报告中对应的位置。

第 1 章

实验相关说明

我们这一组内成员与分工情况如何？实验目的是什么？这一章将对我们本次实验的大致布局作简要介绍。

1.1 组内成员与分工

组内成员信息表如表 1.1。

姓名	学号	班级
马英硕	191220080	周五 5~6 节
许希帆	191870219	周一 5~6 节
张宇晨	191220171	周一 5~6 节

表 1.1 组内成员信息表

由于本次实验工作量较大，我们采取了一定的合理分工形式，并在 GitLab 上建立了合作仓库。网址和更多实验额外信息将在 1.4 节给出。

下面给出组内分工情况：

姓名	主要工作	工作时间
马英硕	SSshell 的外设与 Super I/O SSshell 的硬件拓展功能实现	Maintainer 净工作量超 30 小时
许希帆	SSshell 的 CPU 与 ALU	
张宇晨	SSshell 的软件	

表 1.2 组内成员分工表

1.2 实验目的与预期效果

在我们的实验中，我们力求制造一个用户体验极佳，键盘控制功能和显示器功能强大，程序运行灵活的简单计算机系统。

为了真正实现高质量的用户体验，我们在设计上有多种考虑。下面对整个项目的设计进行介绍。

外设部分

用户在使用 SSshell 时，最直接、主要的体验是使用键盘，并在屏幕上看到显示。

对于键盘，我们希望**尽量实现所有键无冲的完美实现**，并尽可能多地实现实际键盘的功能：

- ✓ 状态键功能；
- ✓ Shift 组合键与 Capslock 键改变输入的符号；
- ✓ **实现小键盘和方向键功能，以及其他二字节扫描码的功能**。

由于键盘上的“Prt Scr”与“Pause Brk”键扫描码太过复杂，在 SSshell 中也没有用武之地，所以我们不实现。

对于显示，我们希望 SSshell 能够像普通电脑一样，开机时有独特画面。所以 **SSshell 也会有独特多彩的开机界面**。

为了实现类似日常使用的控制台程序样式，我们的显示不仅要能支持普通的输入、换行功能，还应该有命令提示符，光标，无限制滚屏等功能。此外，为了良好的用户体验，我们希望能提供多种

控制台配色方案供用户选择，让用户脱离单纯黑底白字的常规界面。

我们希望方向键作为一般来说用途广泛的按键而言，在 SSshell 中也有用武之地。但由于想要实现左右方向键操纵光标和上下键切换指令功能可能会大量消耗板上更多资源，而且与其他操作配套起来十分复杂，所以这里对方向键进行了改造。**左右键可以切换配色方案，上下键可以滚屏查看历史输出记录**。

TODO：增加音频部分和其他部分

1.3 实验环境与器材

- 硬件器材：FPGA 电路板，VGA 显示器，PS2 键盘，耳机。
- 软件环境清单：

软件类型	软件名称
Verilog 编程	Quartus (18.1 Standard & 17.1 Lite)

表 1.3 软件环境清单

1.4 其他实验信息

- GitLab 地址：

  https://git.nju.edu.cn/PandaAwAke/computersystem

  - 如果您需要该代码仓库的许可权，以查看我们的 Commits 记录以及其他仓库记录，请联系马英硕同学。

- 其他参考文档:

✧ 小实验的实验手册

✧ Quartus 使用系列

  ➢ IP 核 https://www.cnblogs.com/mengyi1989/p/11515982.html

✧ MIPS 汇编系列

  ➢ https://www.jianshu.com/p/ac2c9e7b1d8f

  ➢ https://blog.csdn.net/qq_39559641/article/details/89608132

第一部分

外接设备信号处理与接口准备

第一部分介绍

这一部分将深入介绍 SSshell 的外接设备：键盘、VGA 显示器和音频。

第 2 章主要介绍了本实验中键盘有关的设计思想、具体实现和最终效果，第 3 章介绍了在本实验中我们处理 VGA 显示器显示逻辑的方式，以及多种高级显示功能的实现；还展示了如何将 I/O 放在显存中实现。第 4 章介绍了实验中我们对音频部分的处理。

第 2 章

键 盘

键盘是现代计算机操作的最重要输入设备之一，也是计算机用户操作体验中最重要的部分之一。在本章中，我们将深入探讨 SSshell 实现了哪些键盘功能，以及采用了什么方式去实现它们。

2.1 PS2 键盘原理与底层 PS2 模块

PS2 数据信号输出原理

PS/2是个人计算机串行 I/O 接口的一种标准，因其首次在 IBM PS/2（Personal System/2）机器上使用而得名，PS/2 接口可以连接 PS/2 键盘和 PS/2 鼠标。所谓串行接口是指信息是在单根信号线上按序一位一位发送的。

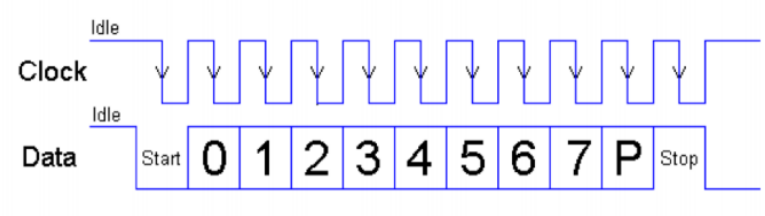


图 2.1 PS2 信号输出数据时序图

PS/2 接口使用两根信号线，一根信号线传输时钟 PS2_CLK，另一根传输数据 PS2_DAT。时钟信号主要用于指示数据线上的比特位在什么时候是有效的。键盘和主机间可以进行数据双向传送，这里只讨论键盘向主机传送数据的情况。当 PS2_DAT 和 PS2_CLK 信号线都为高电平（空闲）时，键盘才可以给主机发送信号。如果主机将 PS2_CLK 信号置低，键盘将准备接受主机发来的命令。在我们的实验中，主机不需要发命令，只需将这两根信号线做为输入即可。

当用户按键或松开时，键盘以每帧 11 位的格式串行传送数据给主机，同时在 PS2_CLK 时钟信号上传输对应的时钟（一般为 10.0–16.7kHz）。第一位是开始位（逻辑 0），后面跟 8 位数据位（低位在前），一个奇偶校验位（奇校验）和一位停止位（逻辑 1）。每位都在时钟的**下降沿**有效，图 8-4显示了键盘传送一字节数据的时序。在下降沿有效的主要原因是下降沿正好在数据位的中间，因此可以让数据位从开始变化到接收采样时能有一段信号建立时间。

键盘通过 PS2_DAT 引脚发送的信息称为扫描码，每个扫描码可以由单个数据帧或连续多个数据帧构成。当按键被按下时送出的扫描码被称为“通码（Make Code）”，当按键被释放时送出的扫描码称为“断码（Break Code）”。以“W”键为例，“W”键的通码是 1Dh，如果“W”键被按下，则 PS2_DAT 引脚将输出一帧数据，其中的 8 位数据位为 1Dh，如果“W”键一直没有释放，则不断输出扫描码 1Dh 1Dh . . . 1Dh，直到有其他键按下或者“W”键被放开。某按键的断码是 F0h 加此按键的通码，如释放“W”键时输出的断码为 F0h 1Dh，分两帧传输。

多个键被同时按下时，将逐个输出扫描码，如：先按左“Shift”键（扫描码为 12h）、再按“W”键、放开“W”键、再放开左“Shift”键，则此过程送出的全部扫描码为：12h 1Dh F0h 1Dh F0h 12h。

键盘扫描码

每个键都有唯一的通码和断码。键盘所有键的扫描码组成的集合称为扫描码集。共有三套标准的扫描码集，所有现代的键盘默认使用第二套扫描码。图 8-5显示了键盘各键的扫描码（以十六进制表示），如 Caps 键的扫描码是 58h。由图 8-5可以看出，键盘上各按键的扫描码是随机排列的，如果想迅速的将键盘扫描码转换为 ASCII 码，一个最简单的方法就是利用查找表 (LookUp Table, LUT)。

所有的键盘扫描码由图 2.2 以及图 2.3 展示。

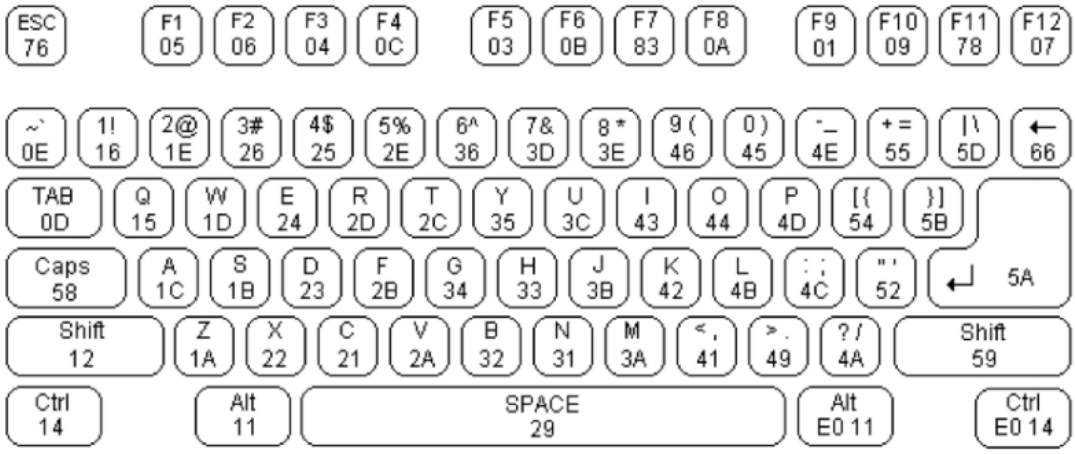


图 2.2 键盘扫描码 (主键盘)

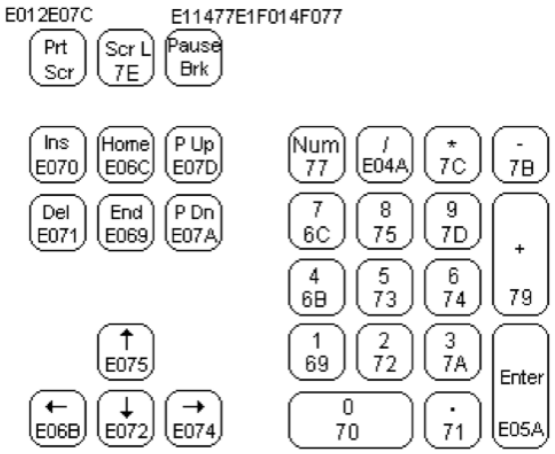


图 2.3 键盘扫描码 (扩展键盘与数字键盘)

在实验 8 中，我们有一个底层键盘处理接口模块：ps2_keyboard。这里不再赘述 ps2_keyboard 的具体内容。该模块将作为 SSshell 最底层处理 PS2 信号的模块。

2.2 实现基础键盘功能

本节主要介绍 SSshell 的键盘处理机制，这些机制被集中在 KeyboardHandler 模块中。

2.2.1 基础键盘状态处理设计

键盘是可以同时按下多个按键的，所以我们在处理键盘时，也希望键盘的状态可以反映出目前所有的按键状态。所以，我们希望每一个键都拥有自己的状态机。单键对应的状态机如图 2.4 所示。

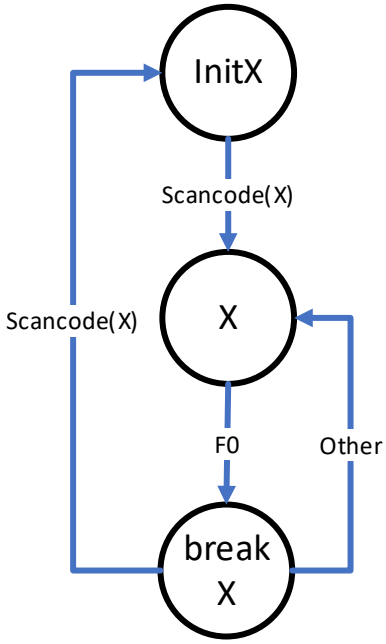


图 2.4 单个键对应的状态机

注：上图中的 X 是一个按键代表。实际上，X 代表了所有的键（在 SSshell 中，我们用扫描码指代一个单字节扫描码键）。这些单键状态机互不干扰，但是**共用同一个断码信号 F0**。

我们可以在接收到断码信号 F0 后，做一个断码标记 breaking，这标记了所有的单键状态机进入 breakX 状态；然后在接收到下一个扫描码时，让该扫描码对应的单键状态机进行状态转移，从 breakX 状态转移到 InitX 状态。

考虑到单字节扫描码一共有 256 种可能（虽然实际上不会出现这么多种），我们可以采用一个位宽为 256 的寄存器作为全体的单字节扫描码单键状态机的状态集合。由于我们用断码标记 breaking 标记了所有键是否处于 breakX 状态，于是我们可以仅仅用 0 和 1 完成对单键状态机的表示。0 代表该键未处于按下状态，1 代表该键正处于被按下的状态。

2.2.2 实现基础键盘处理功能

在 2.2.1 节中，我们已经详述了单键状态机如何实现。但键盘处理远远没有这么简单，我们需要与 ps2_keyboard 对接，需要实现单键状态机，还需要让键盘输出与 ASCII 码相关的信息。在这一部分中，我们将分步骤展示具体的代码。

与 ps2_keyboard 底层模块对接

我们只需要与 ps2_keyboard 内置的 FIFO 队列实现信息传输与对接即可。在此展示对接框架代码：

```
ps2_keyboard inputer(
    .clk(clk),
    .clrn(clrn),
    .ps2_clk(PS2_CLK),
    .ps2_data(PS2_DAT),
    .data(data),
    .ready(ready),
    .nextdata_n(nextdata_n)
);

//=============================================================
//  Clock Logical coding
//=============================================================

always @(posedge clk) begin
    if (nextdata_n == 0) begin  // 让nextdata_n保持一个周期的0
        nextdata_n <= 1;
    end else
        if (ready) begin

            // Keyboard Handling.

            nextdata_n <= 0;
        end
end
```

图 2.5 与 ps2_keyboard 对接

单字节扫描码单键状态机的实现

```
always @(posedge clk) begin
    if (nextdata_n == 0) begin  // 让nextdata_n保持一个周期的0
        nextdata_n <= 1;
    end else
        if (ready) begin
            if (data == 8'hF0) begin
                // 改变状态
                breaking <= 1;
            end else begin
                // 要么从InitX到X，要么从breakX到InitX
                if (breaking) begin          // 接收到F0
                    // 从breakX到InitX
                    breaking <= 0;
                    state[data] <= 0;
                    scanCode <= 0;
                end else begin
                    // 从InitX到X
                    state[data] <= 1;
                    scanCode <= data;
                end
            end
            nextdata_n <= 0;
        end
end
```

图 2.6 基础单键状态机实现

上述红框对应的就是图 2.5 中 Keyboard Handling 部分。

✧ state 是一个位宽为 256 的寄存器，用单键的扫描码作为该键的下标（我们初始化 state 为 0，但这里不进行展示）；

✧ scanCode 是本模块向外界模块**输出**的**有效**键盘扫描码，位宽为 8。

朴素的 ASCII 输出实现

接下来，我们叙述如何输出一个有效按键是否为 ASCII 按键；如果是，输出对应的 ASCII 码。注意，这里只叙述**朴素的** ASCII 输出实现，如何考虑与状态键组合，将在 2.3 节中详述。

首先，增加输出接口：isASCIIkey（1 位寄存器），代表输出一个扫描码时，是否为 ASCII 键。它的实现也非常容易：

```
assign isASCIIkey = (
    scanCode    != 8'h00 && // 无效扫描码
    scanCode    != 8'h0D && // TAB
    scanCode    != 8'h76 && // ESC
    scanCode    != 8'h58 && // CapsLock
    scanCode    != 8'h12 && // LShift
    scanCode    != 8'h14 && // LCtrl
    scanCode    != 8'h11 && // LAlt
    scanCode    != 8'h59 && // RShift
    scanCode    != 8'h66 && // 退格键
    scanCode    != 8'h5A && // LEnter
    scanCode    != 8'h7E && // Scr LK
    ((scanCode > 8'h0C && scanCode != 8'h78) || scanCode == 8'h08) // F1~F12
);
```

图 2.7 isASCIIkey 的输出

实现思路是排除那些不是 ASCII 字符的键。

接下来，我们需要引入 LUT (LookUp Table)，来实现一对一的扫描码-ASCII 字符转换。由于我们这里只阐述朴素 ASCII 输出，所以不考虑进一步处理。我们手里已经有一份转换表（scancode.mif)。

首先，利用 IP Catalog 生成单口 ROM，取消输出 q 的缓冲，如图 2.8 所示。之后，只需要简单连接该 ROM 即可。

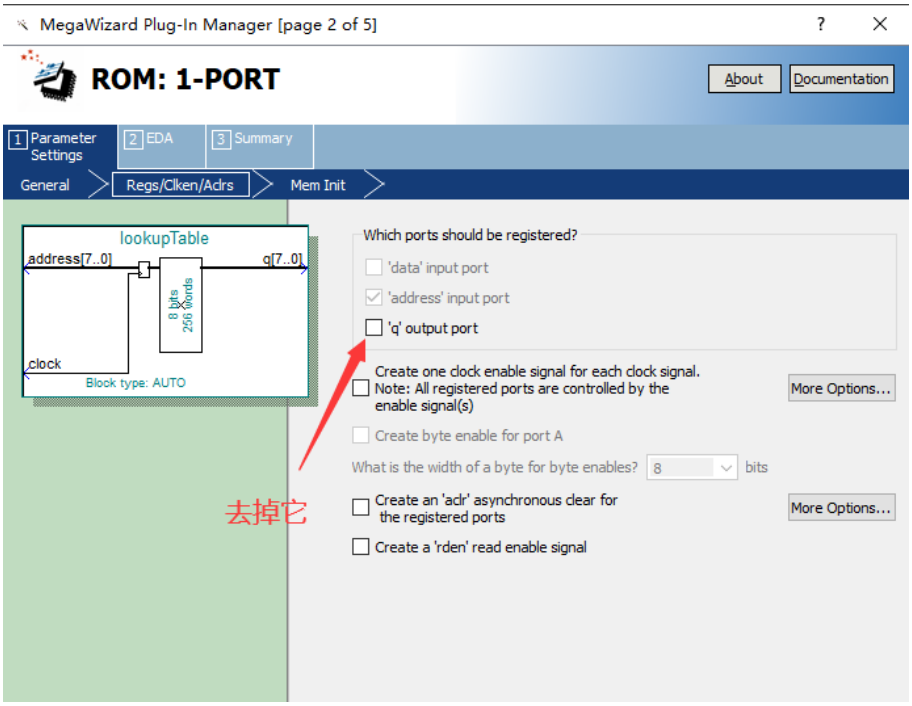


图 2.8 取消输出 q 的缓冲

```
wire [7:0] LUT_ASCII;
lookupTable LUT(
    .address(scanCode),
    .clock(clk),
    .q(LUT_ASCII)
);
```

图 2.9 扫描码转换为朴素 ASCII 码

至此，我们已经实现了最基础的键盘功能。进阶的键盘功能，我们将在后面的小节中讨论。

2.3 功能键处理

在这个小节中，我们将展示如何妥善地向外输出功能键状态，并让它与 ASCII 码配合，实现现代计算机的**全字符输出功能**。

2.3.1 状态跟踪

首先讨论如何输出那些保持按下时有效的状态键，比如 Ctrl、Shift 和 Alt 等。我们引入新的状态键输出接口（不展示），然后对它们进行直接的 assign，思路是把状态键对应的扫描码作为下标去读取 state 数组并输出。注意，这里暂未讨论双字节扫描码，所以 Right Ctrl 和 Right Alt 暂时不在讨论范围内。由于左右 Shift 键都是单字节的扫描码，所以可以直接实现。

```
assign shift = state[18] | state[89];
assign ctrl  = state[20];
assign alt   = state[17];
```

图 2.10 基础状态键状态输出

实现 Capslock 状态

大写锁定键是一个略微特殊的键：在第一次按下时，开启锁定模式；在第二次按下时，关闭锁定模式。我们需要进行加工。

对于 Capslock 键的实现而言，我们认为对于单次按下/松开而言是无差别的。所以我们统一在松开 Capslock 键时处理锁定逻辑。

```
assign capslock = state[88] | capslockflag;

if (breaking) begin          // 接收到F0
    // 从breakX到InitX
    if (data == 8'h58) begin // 大写锁定
        capslockflag <= ~capslockflag;
    end
    breaking <= 0;
    state[data] <= 0;
    scanCode <= 0;
```

图 2.11 大写锁定逻辑

2.3.2 实现 Capslock 的大写转换功能

我们首先需要修改 ASCII 的输出接口，根据 capslock 的状态（实际上是输出端寄存器 capslock 的值），选择进行 Capslock 锁定情况下的处理。图 2.12 展示如何进行这些处理：

```
assign ASCII = (
    (capslock == 1) ?
    capslockCase(LUT_ASCII) :
    LUT_ASCII
);

function [7:0] capslockCase;
    input [7:0] rawCase;
    begin
    if (rawCase >= 8'h61 && rawCase <= 8'h7A)
        capslockCase = rawCase - 8'h20;
    else
        capslockCase = rawCase;
    end
endfunction
```

图 2.12 大写锁定状态下对 ASCII 的处理

由于单纯的大写锁定状态下，只会把小写字母转换为大写字母，所以只有原来的 ASCII 码处于'a'~'z'的区间内时，才将其减去 0x20 作为对应的大写字母 ASCII 码。

2.3.3 实现 Shift 的字符转换功能

在处理 Shift 按键对 ASCII 的影响时，我们要考虑三个问题：

- ✧ 把小写字母转换为大写字母；
- ✧ 按下某些键时，转换为对应的 Shift 按键下的特殊符号；
- ✧ Capslock 模式下，把大写字母转换为小写字母。

首先，我们仍然需要修改 ASCII 的输出。注意，由于 Capslock 键对 ASCII 的处理不依赖于 Shift 键而反过来并非如此，所以处理顺序至关重要。

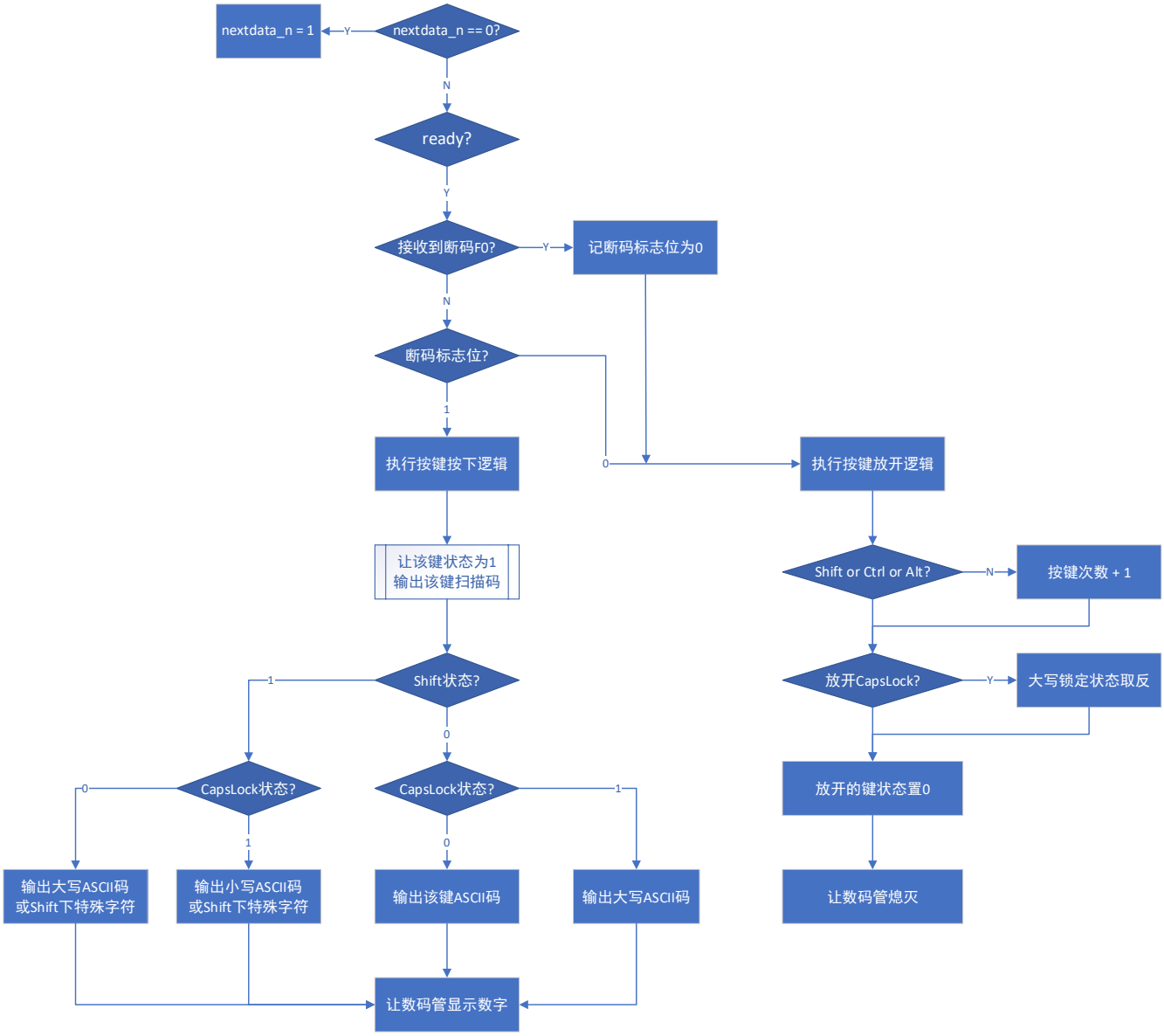
```
assign ASCII = (
    (shift && (scanCode != 0)) ?
    shiftCase(LUT_ASCII, capslock) :
    (
        (capslock == 1) ?
        capslockCase(LUT_ASCII) :
        LUT_ASCII
    )
);

function [7:0] shiftCase;
    input [7:0] rawCase;
    input capslock;
    begin
    if (rawCase >= 8'h61 && rawCase <= 8'h7A)
        if (capslock == 0)
            shiftCase = rawCase - 8'h20;
        else
            shiftCase = rawCase;
    case (rawCase)  // 符号表
        8'h60: shiftCase = 8'h7E; 8'h31: shiftCase = 8'h21; 8'h32: shiftCase = 8'h40;
        8'h33: shiftCase = 8'h23; 8'h34: shiftCase = 8'h24; 8'h35: shiftCase = 8'h25;
        8'h36: shiftCase = 8'h5E; 8'h37: shiftCase = 8'h26; 8'h38: shiftCase = 8'h2A;
        8'h39: shiftCase = 8'h28; 8'h30: shiftCase = 8'h29; 8'h2D: shiftCase = 8'h5F;
        8'h3D: shiftCase = 8'h2B; 8'h5C: shiftCase = 8'h7C; 8'h5B: shiftCase = 8'h7B;
        8'h5D: shiftCase = 8'h7D; 8'h3B: shiftCase = 8'h3A; 8'h27: shiftCase = 8'h22;
        8'h2C: shiftCase = 8'h3C; 8'h2E: shiftCase = 8'h3E; 8'h2F: shiftCase = 8'h3F;
    endcase
    end
endfunction
```

图 2.13 Shift 状态下对 ASCII 的处理

图中的符号表部分，是 Shift 状态下一些 ASCII 码对应的转换后的 ASCII 码，比如如果原来的 ASCII 字符是'/'，将被转换为'?'。另外，shiftCase 需要接收 Capslock 状态并进行处理，如果 Capslock 状态也为 1，则输出原来的小写字符。

在本节的最后，给出截至本节的键盘实现流程图，便于读者理解。



2.4 实现双字节扫描码支持

我们希望键盘能够实现尽可能多的功能，而键盘上还有很多键的扫描码是以 E0 开头的双字节扫描码。在这一节中，我们丰富 KeyboardHandler，使其支持双字节扫描码，这样我们就实现了键盘上除了“Prt Scr”与“Pause Brk”以外的全键识别。

要注意的是，支持双字节扫描码后，我们需要做大量的支持修改，也可以添加许多新的功能，比如：

- ✧ 双字节扫描码状态机实现；
- ✧ isASCIIkey 和输出的 ASCII 维护与修改；
- ✧ 右 Ctrl 和右 Alt 的状态键支持；
- ✧ Insert、Delete、Home、End 等键以及小键盘斜杠支持；
- ✧ 方向键支持；
- ✧ ……

接下来，我们展示如何一步步具体实现它们。

双字节扫描码状态机

我们只需要模仿之前的 state、scanCode、breaking 机制，定义新的寄存器 state_E0、scanCode_E0 和 preE0，其中 preE0 代表前一个键盘信号字节是 E0。state_E0、scanCode_E0 对应的是 E0 后的那个字节。

```
if (ready) begin
    if (data == 8'hF0) begin
        // 改变状态
        breaking <= 1;
        preE0 <= 0;
    end else if (data == 8'hE0) begin
        preE0 <= 1;
    end else begin
        // 要么从InitX到X，要么从breakX到InitX
        if (breaking) begin          // 接收到F0
            if (preE0) begin             // 也接收过E0
                // 这里接受F0 E0后的那个扫描码
                breaking <= 0;
                preE0 <= 0;
                state_E0[data] <= 0;
                scanCode_E0 <= 0;
            end else begin               // 只接收到F0，没接收到E0
                // 从breakX到InitX
                if (data == 8'h58) begin // 大写锁定
                    capslockflag <= ~capslockflag;
                end
                breaking <= 0;
                state[data] <= 0;
                scanCode <= 0;
            end
        end else begin
            if (preE0) begin             // 前一个是E0
                preE0 <= 0;
                state_E0[data] <= 1;
                scanCode_E0 <= data;
                scanCode <= 0;
            end else begin
                // 从InitX到X
                state[data] <= 1;
                scanCode <= data;
                scanCode_E0 <= 0;
            end
        end
    end
    nextdata_n <= 0;
end
```

图 2.14 双字节扫描码状态机

我们可以在非 breaking 状态下，检测 E0 的输入，然后模拟 breaking 维护一个 preE0 状态。要注意：接收到 F0 后可能会接收到 E0，此时不会进入红框部分，而是 breaking 和 preE0 标记均为 1；在红框部分，我们处理 breaking 和 preE0 的四种组合状态，处理方式如下表：

breaking	preE0	操作
0	0	更新 state 状态机，输出 scanCode
0	1	更新 state_E0 状态机，输出 scanCode_E0
1	0	更新大写锁定和 state，清空输出
1	1	更新 Insert(见下文)和 state_E0，清空输出

表 2.1 双字节扫描码处理操作

要注意的是，我们不能同时向外输出 scanCode 和 scanCode_E0，否则外界模块将不知道键盘按下的是哪一个键。所以在设计中，**我们将时刻保持两者只有一个不为 0 (无效输出)**。

isASCIIkey 与输出的 ASCII 修改

```
assign isASCIIkey = (
    (   scanCode != 8'h00 &&
        scanCode != 8'h0D && // TAB
        scanCode != 8'h76 && // ESC
        scanCode != 8'h58 && // CapsLock
        scanCode != 8'h12 && // LShift
        scanCode != 8'h14 && // LCtrl
        scanCode != 8'h11 && // LAlt
        scanCode != 8'h59 && // RShift
        scanCode != 8'h66 && // 退格键
        scanCode != 8'h5A && // LEnter
        scanCode != 8'h7E && // Scr LK
        ((scanCode > 8'h0C && scanCode != 8'h78) || scanCode == 8'h08) // F1~F12
    ) ||
    (scanCode_E0 == 8'h4A) // 右边的除号是，其他的E0开头的都不是
);
```

图 2.15 更新 isASCIIkey

由于 E0 开头的双字节扫描码的键中，只有小键盘的斜杠是 ASCII 字符，所以我们只需要在检测中针对双字节扫描码检测斜杠即可。

此外，ASCII 输出也需要做略微修改：

```
assign ASCII_helper = (
    (scanCode != 0) ?
    LUT_ASCII :
    8'h2F       // 右边小键盘的斜杠
);
assign ASCII = (
    (shift && (scanCode != 0)) ?
    shiftCase(ASCII_helper, capslock) :
    (
        (capslock == 1) ?
        capslockCase(ASCII_helper) :
        ASCII_helper
    )
);
```

图 2.16 更新 ASCII 输出

状态键的完美实现

现在，我们成功支持了双字节扫描码识别，所以我们可以对一些之前的状态键进行维护，也可以加入新的 Insert 状态：

```
if (breaking) begin          // 接收到F0
    if (preE0) begin             // 也接收过E0
        // 这里接受F0 E0后的那个扫描码
        if (data == 8'h70) begin // Insert模式
            insertflag <= ~insertflag;
        end
assign ctrl =   state[20] | state_E0[20];
assign alt =    state[17] | state_E0[17];
assign insert = state_E0[112] | insertflag;
```

图 2.17 双字节扫描码状态键

这里的 insert 状态实现和之前的 Capslock 原理类似，不再赘述。

至此，我们完成了 KeyboardHandler 对双字节扫描码处理的职责。至于外界如何识别并使用这些按键（比如方向键等），在这里并不关心。接下来，还有一个重要的问题：**外界如何知道有一个新的有效扫描码产生了呢？**

2.5 对外接口：实现有效键信号向外传递

在前面的小节中，我们已经实现了 KeyboardHandler 的各种高级功能。键盘的一个键如果被按下，或始终未被松开，将一直发送有效扫描码。实际上，对于本实验而言，识别断码并向外传递断码信号没有价值（本实验对键松开识别没有要求），所以 KeyboardHandler 只需要告诉外界模块：有新的扫描码产生了。

- 注意，我们不能单纯从外界检测扫描码是否有变化来确定这一点，否则逻辑非常复杂而且按下不松开时将无法检测！
- 设计思路是：增加一个输出端口 newKey，当外界检测到 newKey 的上升沿时，代表有新的有效扫描码产生！

我们的思路是，增加一个缓冲变量 buffer_newKey（位宽 3）去缓冲一个新键产生的信号，然后让 newKey 始终为 buffer_newKey[2]即可。之所以设计缓冲机制，是因为如果 newKey 出现了上升沿，此时距有效扫描码产生一定过去了 2 个时钟周期；**我们希望外界读到 newKey 的上升沿时，读入的扫描码一定是正确的**。由于两个有效扫描码产生的间隔至少需要十几个时钟周期，所以这种缓冲是可行的！

```
assign newKey = buffer_newkey[2];
```

```
always @(posedge clk) begin
    if (nextdata_n == 0) begin  // 让nextdata_n保持一个周期的0
        nextdata_n <= 1;
        buffer_newkey <= {buffer_newkey[1:0], 1'b0};
    end else
        if (ready) begin
            if (data == 8'hF0) begin
                ...
                buffer_newkey <= {buffer_newkey[1:0], 1'b0};
            end else if (data == 8'hE0) begin
                ...
                buffer_newkey <= {buffer_newkey[1:0], 1'b0};
            end else begin
                if (breaking) begin
                    if (preE0) begin
                        ...
                        buffer_newkey <= {buffer_newkey[1:0], 1'b0};
                    end else begin
                        ...
                        buffer_newkey <= {buffer_newkey[1:0], 1'b0};
                    end
                end else begin
                    if (preE0) begin
                        ...
                        buffer_newkey <= {buffer_newkey[1:0], 1'b1};
                    end else begin
                        ...
                        buffer_newkey <= {buffer_newkey[1:0], 1'b1};
                    end
                end
            end
            nextdata_n <= 0;
        end else
            buffer_newkey <= {buffer_newkey[1:0], 1'b0};
end
```

图 2.18 buffer_newKey

至此，我们完成了 KeyboardHandler 的全部功能。它已经成为了一个比较高级的键盘处理模块。

第 3 章

VGA 与 I/O

显示器，是现代计算机最重要的输出设备之一。在本章中，我们将介绍 SSshell 的显存处理机制：包括如何实现多功能的控制台字符回显功能，以及如何与外部模块(CPU)对接实现 I/O。这些机制被封装在 VideoMemory 模块中。

3.1 VGA 原理与字符显示原理

VGA 显示信号原理

VGA（Video Graphics Array）接口，即视频图形阵列。VGA 接口最初是用于连接 CRT 显示器的接口，CRT 显示器因为设计制造上的原因，只能接受模拟信号输入，这就需要显卡能输出模拟信号。关于模拟信号和数字信号的区别，请参考 (Analog Signal及Digital Signal)。VGA 接口就是显卡上输出模拟信号的接口，在传统的 CRT 显示器中，使用的都是 VGA 接口，现在仍有不少液晶显示器或投影仪还支持 VGA 口。VGA 接口是 15 针/孔的梯形插头，分成 3 排，每排 5 个，如图 3.1 所示：

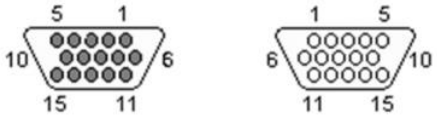


图 3.1 VGA 接口

VGA 接口的接口信号主要有 5 个：R（Red）、G（Green）、B（Blue）、HS（Horizontal Synchronization）和 VS（Vertical Synchronization），即红、绿、蓝、水平同步和垂直同步（也称行同步和帧同步）。

图像的显示是以像素（点）为单位，显示器的分辨率是指屏幕每行有多少个像素及每帧有多少行，标准的 VGA 分辨率是 640×480，也有更高的分辨率，如 1024×768、1280×1024、1920×1200 等。从人眼的视觉效果考虑，屏幕刷新的频率（每秒钟显示的帧数）应该大于 24，这样屏幕看起来才不会闪烁，VGA 显示器一般的刷新频率是 60HZ。

每一帧图像的显示都是从屏幕的左上角开始一行一行进行的，行同步信号是一个负脉冲，行同步信号有效后，由 RGB 端送出当前行显示的各像素点的 RGB 电压值，当一帧显示结束后，由帧同步信号送出一个负脉冲，重新开始从屏幕的左上端开始显示下一帧图像，如图 3.2 所示。

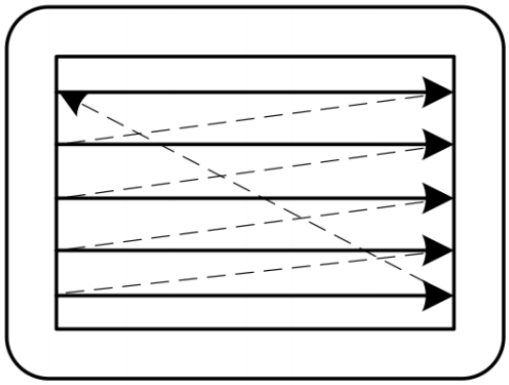


图 3.2 扫描器扫描示意图

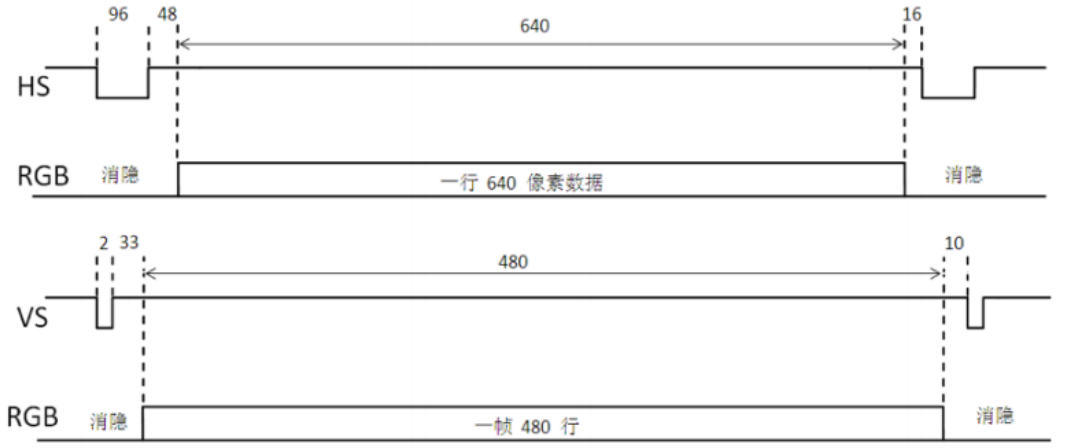


图 3.3 VGA 行扫描、场扫描时序示意图

RGB 端并不是所有时间都在传送像素信息，由于 CRT 的电子束从上一行的行尾到下一行的行头需要时间，从屏幕的右下角回到左上角开始下一帧也需要时间，这时 RGB 送的电压值为 0（黑色），这些时间称为电子束的行消隐时间和场消隐时间，行消隐时间以像素为单位，帧消隐时间以行为单位。VGA 行扫描、场扫描时序示意图如图 3.3 所示：

由图 3.3 可知，在标准的 640 × 480 的 VGA 上有效地显示一行信号需要 96+48+640+16=800 个像素点的时间，其中行同步负脉冲宽度为 96 个像素点时间，行消隐后沿需要 48 个像素点时间，然后每行显示 640 个像素点，最后行消隐前沿需要 16 个像素点的时间。所以一行中显示像素的时间为 640 个像素点时

间，一行消隐时间为 160 个像素点时间。

在标准的 640×480 的 VGA 上有效显示一帧图像需要 2+33+480+10=525 行时间，其中场同步负脉冲宽度为 2 个行显示时间，场消隐后沿需要 33 个行显示时间，然后每场显示 480 行，场消隐前沿需要 10 个行显示时间，一帧显示时间为 525 行显示时间，一帧消隐时间为 45 行显示时间。

因此，在 640×480 的 VGA 上的一幅图像需要 525×800 =420k 个像素点的时间。而每秒扫描 60 帧共需要约 25M 个像素点的时间。

字符显示原理

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，如图 3.4 所示。

图 3.4 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。同学们可以自己用高级语言生成点阵存储文件。我们也提供了可通过 $readmemh 语句读取的点阵文本文件，其中每 3 个 16 进制数 (共 12bit) 表示单个字符的一行，该行的 9 个点中的最左边点在 12bit 中的最低位（**请注意高低位顺序**），然后依次类推，最高的 3 个 bit 始终为 0。每个字符 16 行，共 256 个字符。

例如，ASCII 字符“A”的编码是 41h（十进制 65）。因此其字模对应的地址是 16 × 65 = 1040（文本文件起始从 1 行开始，因此在第 1041 行）。以 A 字符的第 4 行为例，文件中存储的是 038h，二进制对应是 0000 0011 1000。最低位为 0，所以左边第一个像素为 0，左起第 4 到第 6 个像素为 1。如图 3.5 所示，此处为方便显示颜色是黑白颠倒的。

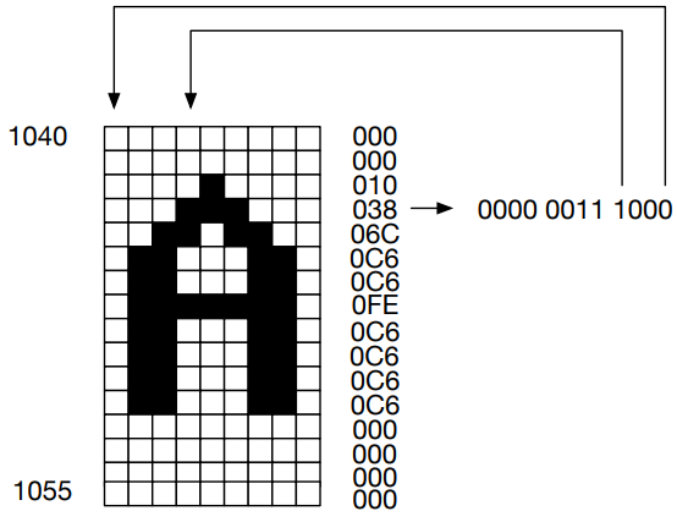


图 3.5 字模“A”与存储器的关系

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。这样，我们的字符显存只需要 2.1kByte，加上点阵的 6.144kByte，总共只需要不到 10kByte 的存储，FPGA 片上的存储足够实现了。

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 $(0 \le x \le 69, 0 \le y \le 29)$。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。

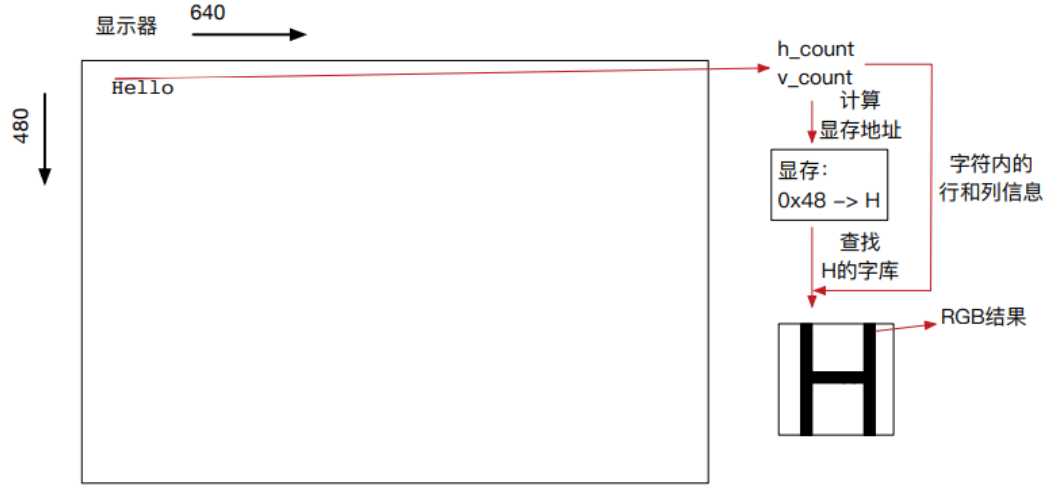


图 3.6 字符显示流程示意图

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。此处可以显示黑底白字或其他彩色字符，只需要按自己的需求分别设置背景颜色和字符颜色即可。

3.2 实现 SSshell 的基础字符回显系统

3.2.1 显示字符内容存储

想要实现对屏幕上字符的操作，我们需要维护寄存器中的字符序列。另外，如果需要显示到屏幕上，还需要存储每个字符的像素显示信息。

我们采用寄存器 keys（位宽 8，长度 4200）存储所有的字符，用 cursor 作为光标变量指向下一个写入位置。由于一个屏幕可以显示 $30 * 70 = 2100$个字符，而 keys 拥有 4200 个位置，所以我们可以记录两屏幕的字符。

最容易说明的是，我们的字符像素显示存储器可以直接采用 IP 核双口 ROM 的方式实现（为什么是双口的将在后面解释）。

```
vga_memory vMemoryROM(
    .address_a(vm_index),
    .address_b(vm_index_header),
    .clock(clk),
    .q_a(line),
    .q_b(line_header)
);
```

图 3.7 字符像素显示存储器

接下来，我们结合具体操作来分析如何进行对 keys 的写入/读取。至于如何最终将它们显示到屏幕上，请查看 3.2.2 节。

字符写入

首先考虑把字符写入 keys 的操作。要注意，我们针对 keys 的操作可能非常复杂，可能会写入一个有效字符或删除一个字符（即写入 0）。所以 keys 一定是一个双口 RAM。如果符合 RAM 的操作规范，我们在一个时钟沿只能对其最多两个下标进行写入/读取，否则可能会在综合时产生严重问题（如下），我们组内戏称其为“**织毛衣现象**”：

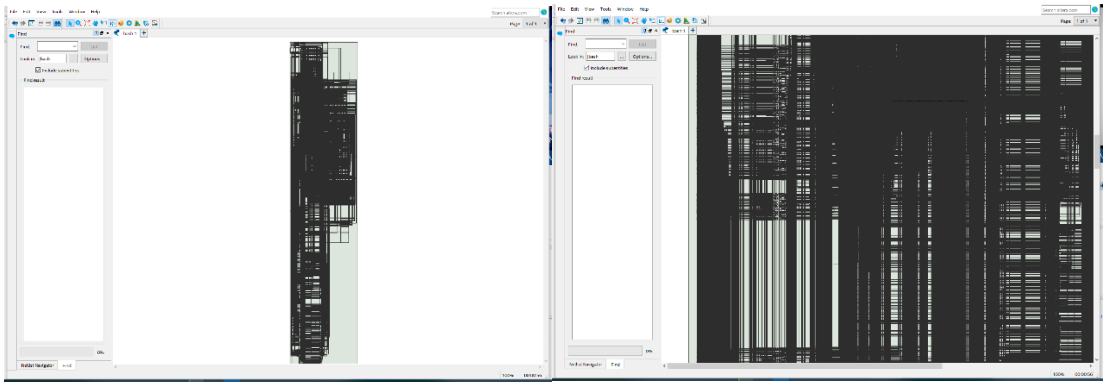


图 3.8 “织毛衣现象”

我们的设计逻辑和代码逻辑在**实际上是不会在同一个时钟沿对 keys 的多个下标进行写入的**，但 Quartus 可能**很难识别**这些互相排斥的逻辑。为了达到目的，而且让 Quartus 更容易综合，我们不妨规定 keys 的写入规则：

✧ 采取缓冲机制，**要写入时在下一个时钟沿写入：**

- 用标识寄存器：flag_keys_write 来标记下一个时钟沿是否需要写入；
- 用 keys_index_helper 记录将要写入的位置下标；
- 用 keys_ASCII_help 记录将要写入的内容。

✧ 原来的写入逻辑：$keys[cursor] <= ASCII;$ 就变成了：

```
flag_keys_write <= 1;
keys_index_helper <= cursor;
keys_ASCII_help <= ASCII;
```

图 3.9 keys 的缓冲写入机制

✧ 在下一个时钟沿，我们只需要进行如下操作：

```
if (flag_keys_write) begin    // 缓存机制：keys在下一个周期进行存储
   keys[keys_index_helper] <= keys_ASCII_help;
   keys_index_helper <= 0;
   flag_keys_write <= 0;
   keys_ASCII_help <= 0;
end
```

图 3.10 keys 的缓冲写入机制

由于非阻塞赋值的性质，我们在这个时钟沿仍然可以更改上述三个辅助缓冲变量。所以实际上，在连续写入时每个上升沿都会对 keys 进行写入，但在代码逻辑上变得明显：写入时仅有 keys_index_helper 一个下标。这样 Quartus 就非常容易综合了。

此外，在标准命令行中，经常会出现输入字符的数量超过一行能包含的上限的情况，这时需要进行换行。我们用变量 x_cnt 记录当前一行的字符数量（范围：0~69），用变量 y_cnt 记录当前已经用到的行（范围：0~59，因为 keys 一共可以存 60 行）以处理自动换行的问题。目前 y_cnt 是一个用不到的变量，但在后面的部分我们将看到它的作用。

有了上述铺垫，我们来看看如何写入一个普通 ASCII 字符：

```
if (scanCode != 8'h66 && isASCIIkey) begin   // 其他正常字符键

   //keys[cursor] <= ASCII; 缓冲输入
   flag_keys_write <= 1;
   keys_index_helper <= cursor;
   keys_ASCII_help <= ASCII;

   cursor <= cursor + 1;
   // 处理x_cnt和y_cnt
   if (x_cnt == 69) begin
      y_cnt <= y_cnt + 1;
      x_cnt <= 0;
   end else begin
      x_cnt <= x_cnt + 1;
   end
end
```

图 3.11 基础的字符写入功能

字符删除

字符的删除，实际上就是在 cursor-1 处写入 0。它的代码也非常简单：

```
// keys[cursor - 1] <= 0;
flag_keys_write <= 1;
keys_index_helper <= cursor - 1;
keys_ASCII_help <= 0;

// 处理x_cnt和y_cnt
x_cnt <= x_cnt - 1;
cursor <= cursor - 1;
```

图 3.12 基础的字符删除功能

3.2.2 显示字符显示功能

首先要说明的是，VideoMemory 与底层模块 vga_control 是**并列**关系。VideoMemory 的功能**仅仅是从 vga_control 处获取坐标信息，并向 vga_control 提供颜色值**！

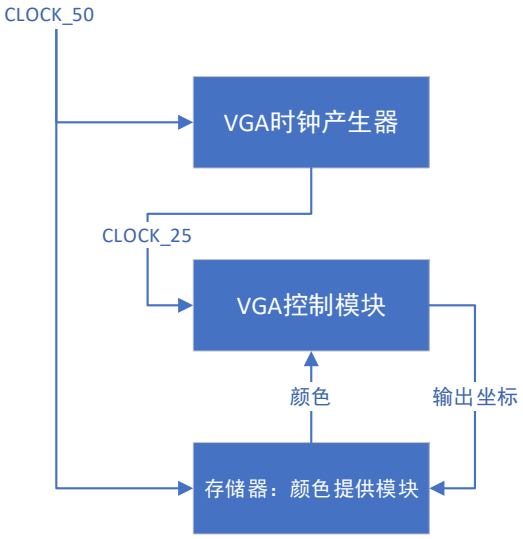


图 3.13 VGA 显示主逻辑

所以，我们只要根据 3.2.1 中的 keys 和字符像素显示存储器，结合 vga_control 提供的 h_addr 和 v_addr 计算出我们要传递的颜色值即可。

由于 FPGA 板计算乘除法会及其繁琐，我们这里采用了以**读 ROM 代乘除法**的方式完成复杂计算。

图 3.14 颜色输出逻辑

这部分逻辑可能比较复杂。我们做进一步解释：

- h_addr 的范围是 0~639，由于做除法比较费时间，我们采用查表方式：
  - KeysXROM 的第 0~8 号成员：0
  - KeysXROM 的第 9~17 号成员：1
  - ……
  - KeysXROM 的第 621~629 号成员：69
  - KeysXROM 的第 630~638 号成员：70，但不会使用
  - KeysXROM 的第 639 号成员：71，但不会使用
- v_addr 的范围是 0~479，由于做除法比较费时间，我们采用查表方式：
  - KeysYROM 的第 0~15 号成员：0
  - KeysYROM 的第 16~31 号成员：1
  - ……
  - KeysYROM 的第 464~479 号成员：29
- keys_baseROM 利用算出的 keysY 算出该行行首对应的 keys 下标位置：
  - keys_baseROM 的第 0 号成员：0
  - keys_baseROM 的第 1 号成员：70
  - keys_baseROM 的第 2 号成员：140
  - ……
  - keys_baseROM 的第 59 号成员：4130

经过上述所有运算，我们可以用如下表达式计算出 keys_index 获取到(h_addr, v_addr)对应的 ASCII 字符：

```
assign keys_index = keys_base_out + keysX;
assign showASCII = keys[keys_index];
```

图 3.15 颜色输出逻辑-ASCII 获取

然后可以用如下表达式计算出颜色值：

```
// 应该显示的ASCII位置
assign offsetX = h_addr - baseX_out;
assign offsetY = v_addr - baseY_out;
assign vm_index = ASCII_base_out + offsetY;
assign showcolor = line[offsetX] ? color_text : color_background;
```

图 3.16 颜色输出逻辑-颜色输出

其中的 color_text 和 color_background 是控制台的文本颜色和背景颜色（24 位）。目前，color_text 是 24'hFFFFFF，color_background 是 24'h0。但 showcolor 还不是最终要显示的颜色，因为可能有(h_addr, v_addr)对应不到的字符（比如横坐标超过 630）。我们需要再次加工。

```
always @(negedge clk) begin
    if (h_addr >= 630)
        rgb <= color_background;
    else
        rgb <= showcolor;
end
```

图 3.17 颜色输出逻辑-颜色加工

注意，上述颜色输出采用了**下降沿**的形式。因为我们采用了上升沿去对 keys 进行操作，而颜色显示与 keys 操作应该是互不干扰的。为了减少逻辑干扰的可能性，我们把显示放在了下降沿。另外，rgb 还有更灵活的作用。我们在后面的小节中可以见到。

至此，rgb 就是(h_addr, v_addr)所对应的颜色值。我们可以将它传递出去，由上层模块将它连接到 vga_control 实现显示功能。

3.2.3 光标

控制台程序中的光标是一个必不可少的显示部分。由于上一节中我们设计的机制比较灵活，所以显示光标的逻辑变得比较简单。

首先，我们需要让光标具有闪烁的功能。考虑到时钟周期需要是

50M 的因子，我们让时钟周期为 2，即一秒钟闪烁两次。

```
// 光标显示使能端
clkgen #(2) cursorclk(
    .clkin(clk),
    .rst(0),
    .clken(1),
    .clkout(cursor_en)
);
```

图 3.18 光标使能端逻辑

图 3.18 创造了光标使能端，clkgen 的功能是产生指定频率的时钟。接下来只需要改造 rgb 即可：

```
always @(negedge clk) begin
   if (h_addr >= 630) begin
      rgb <= color_background;
   end else if (keys_index == cursor && cursor_en)// 光标部分
      if (insert) begin
         if (offsetY < 11)  // Insert模式，光标高度为5(/16)
            rgb <= showcolor;
         else
            rgb <= color_text;
      end else begin
         if (offsetY < 13)  // 非Insert模式，光标高度为3(/16)
            rgb <= showcolor;
         else
            rgb <= color_text;
      end
end
```

图 3.19 光标显示逻辑

我们知道控制台程序中，Insert 状态和非 Insert 状态光标的高度不一样。SSshell 也体现出了这种差异。另外需要注意的是，keys_index 是要显示的那个字符下标，而 cursor 是光标位置，只有它们相同时，才代表这个位置是光标。

3.2.4 接入 KeyboardHandler

到目前为止，我们都没有引入键盘接入。在 2.5 节中，我们说明了 KeyboardHandler 如何向外传递有效扫描码信号。在这一节中，我

们将用特定方式与 KeyboardHandler 对接。

在 2.5 节中，规定外界模块接入 KeyboardHandler 时需要检测 newKey 的上升沿，来说明有新的扫描码产生。但，我们不能在 VideoMemory 中使用语句 posedge newKey，因为我们的操作可能会很复杂，主要依赖于 CLOCK_50 而并非 newKey；也就是说，在 CLOCK_50 上升沿时应该处理 newKey 的信号。

```
reg [2:0] newKey_sync;
wire sampling_newKey = ~newKey_sync[2] & newKey_sync[1];
initial begin
    newKey_sync = 0;
end
always @(posedge clk) begin
    newKey_sync <= {newKey_sync[1:0], newKey};
end
```

图 3.20 检测 newKey 的上升沿

这样，在后面的操作中，只需要判断 sampling_newKey 是否为 1 即可。这样做的原因是，我们传递给 KeyboardHandler 的时钟与这里的时钟是一样快的，时序上不会出现问题。

3.3 实现 SSshell 的高级显示功能

作为一款高大上的控制台程序，我们肯定要引入更多高端功能。这一节中，我们讨论如何实现 SSshell 的高级显示功能。

3.3.1 命令提示符

首先要设计一下如何实现命令提示符。有两种比较容易想到的方式，一种直接在显示阶段在一些行的行首显示命令提示符，另一种在合适的时机将命令提示符的内容加入字符存储器。虽然后一种在控制

台软件中更为普遍，但在 verilog 中实现起来比第一种情况更复杂。在 SSshell 中，第一种情况已经可以满足需要。所以，我们采用第一种方式。

首先，我们需要给出命令提示符内容。

```
function [7:0] Header;  // 命令提示符内容: SSshell
   input [7:0] index;
   case (index)
      0: Header = 8'h53;
      1: Header = 8'h53;
      2: Header = 8'h73;
      3: Header = 8'h68;
      4: Header = 8'h65;
      5: Header = 8'h6C;
      6: Header = 8'h6C;
      7: Header = 8'h24;
      8: Header = 8'h20;
      default: Header = 0;
   endcase
endfunction
```

图 3.21 命令提示符内容

然后我们只需要模拟 showcolor 的逻辑。在 3.2 节中我们讨论了如何从 keys 中取出要显示的字符 showASCII，但在显示命令提示符的时候只需要从函数 Header 处取字符；之后拿着这个字符去字符像素显示存储器中找显示颜色即可。

```
vga_memory vMemoryROM(
    .address_a(vm_index),
    .address_b(vm_index_header),
    .clock(clk),
    .q_a(line),
    .q_b(line_header)
);

ASCII_base ASCII_baseROM(
    .address_a(showASCII),
    .address_b(Header(keysX)),
    .clock(clk),
    .q_a(ASCII_base_out1),
    .q_b(ASCII_base_out2)
);

// 命令提示符
assign vm_index_header = ASCII_base_out2 + offsetY;
assign showcolor_header = line_header[offsetX] ? color_text : color_background;
```

图 3.22 命令提示符颜色逻辑

在图 3.22 中，line_header 就包含了命令提示符字符中 v_addr 所在的那一行对应的 9 个像素点。我们只需要模拟 showcolor 的逻辑，来得出最终显示：showcolor_header 的颜色。

虽然我们计算出了命令提示符处应该显示的颜色，但是还需要限

制光标的一些动作（比如退格等），另外还需要改变颜色输出逻辑，选择输出的颜色是背景色（$h\_addr \geq 630$）、光标、用户输入字符颜色（keys）还是命令提示符字符颜色。

实际上，命令提示符对光标的影响主要在于退格和回车的操作。这些操作牵扯的功能非常复杂，所以我们在 3.3.3 节深入讨论。在本节中，我们仅展示如何在输出颜色时做选择。

```
parameter BASH_HEAD_LEN = 9;
always @(negedge clk) begin
   if (h_addr >= 630) begin
      rgb <= current_vout_color_background;
   end else if (keys_index == cursor && cursor_en) begin // 光标部分
      if (insert) begin
         if (offsetY < 11)  // Insert模式，光标高度为5(/16)
            rgb <= showcolor;
         else
            rgb <= current_vout_color_text;
      end else begin
         if (offsetY < 13)  // 非Insert模式，光标高度为3(/16)
            rgb <= showcolor;
         else
            rgb <= current_vout_color_text;
      end
   end else if (enter[keysY + roll_cnt_lines] && keysX < BASH_HEAD_LEN) begin // 命令提示符
      rgb <= showcolor_header;
   end else begin // 正常部分
      rgb <= showcolor;
   end
end
```

图 3.23 命令提示符显示输出逻辑

BASH_HEAD_LEN 是命令提示符占据的长度。我们采取这种 parameter 的形式来写代码，方便我们更改命令提示符的内容。

图 3.23 中，(keysX, keysY)是(h_addr, v_addr)处对应的是 keys 里的字符下标（keysX 是横坐标，keysY 是纵坐标）。roll_cnt_lines 是滚屏时所需的一个变量(见 3.3.2 节)，在这里可以认为它不存在。**enter[i]代表行 i 是否有命令提示符**，这应该在输入时进行维护。由于 enter[i]也牵扯了大量功能，所以我们只展示含义，不过多讲述如何维护。

上图的逻辑就是：如果这一行有命令提示符，而且要显示的字符下标在命令提示符范围内，我们选择输出命令提示符的颜色。

3.3.2 滚屏处理

一个控制台程序能存储的字符数是有上限的。一个不能滚屏的控制台程序不是一个好程序。在 SSshell 中，规定了 keys 最多可以存储 57 行字符（虽然空间可以存 60 行），超过 57 行时将舍弃第一行的数据，并所有行前移；而且 SSshell 具有自动滚屏和手动滚屏两种功能。

滚屏设计

当一行太长导致需要换行时，虽然在 3.2 节中叙述的显示逻辑中会在显示屏上自动换行，但我们仍然维护了 y_cnt 变量。这个变量将在滚屏时发挥巨大作用，它记录了目前我们的显示屏上有多少行。

首先我们讨论：在非必须时（指超过 57 行需要清空首行），如何在不改动 keys 的情况下完成滚屏显示？也就是，我们如何设计滚屏？

在 3.2 节中，我们先计算了要显示的字符是屏幕上的第几行第几列。我们可以从纵坐标 keysY 下手，记录滚屏的相关信息，然后让 keysY 加上一个滚屏参数，即可轻而易举地实现滚屏！此后，我们只需要维护滚屏参数即可。

在 SSshell 中，用三个变量去刻画滚屏的特征。

```
// 滚屏记录
reg [7:0]  roll_cnt_lines;          // 滚屏滚掉多少行
reg [12:0] roll_cnt;                // 滚屏滚掉的下标
reg [12:0] roll_cnt_max;            // 滚屏滚掉的下标上限
```

图 3.24 滚屏记录变量

我们需要解释的是，为什么三个变量都要维护？它们分别有什么作用？

✧ roll_cnt_lines：假设屏幕上要显示第 keysY 行字符。加入滚屏机制以前，去判断这一行有没有命令提示符时，需要读 enter[keysY]；此后，需要读 enter[keysY + roll_cnt_lines]；

✧ roll_cnt：假设屏幕上要显示的字符位置是(keysX, keysY)。加入滚屏机制以前，想要读取这个字符对应 keys 的下标是什么，有

```
assign keys_index = keys_base_out + keysX;
```

此后，有

```
assign keys_index = roll_cnt + keys_base_out + keysX;
```

✧ roll_cnt_max：roll_cnt 的上限。这是手动滚屏需要用的，如果不考虑手动滚屏，这个值始终与 roll_cnt 一致。

维护它们的值也很简单，下面给出例子：

```
roll_cnt <= roll_cnt - 70;
roll_cnt_lines <= roll_cnt_lines - 1;
roll_cnt_max <= roll_cnt_max - 70;
```

图 3.25 滚屏记录变量的维护

在需要滚屏的时候采取这种维护方式更改变量的值即可。

但是，如果行数太多，在某一次换行处我们需要清空第一行。这时就需要一个清屏标记：ROLL_CLEAR_FIRST_LINE。**我们规定，在清屏时应该专注于清屏，这个时钟周期不应该处理除了清屏以外的事情，否则可能会导致一连串的各种错误**！这一特点，在 3.4 节的 I/O 接口中也很重要，到 3.4 节我们再讨论。

如何实现清屏呢？我们腾出 4000 多个时钟周期专门处理清屏问题，需要移动寄存器的内部值；在清屏的时候，也需要进行一些其他操作。

下面给出清屏操作的实现：

```
//////////////// Screen Rolling Coding ////////////////////
if (ROLL_CLEAR_FIRST_LINE) begin        // 滚屏到57行了，把后面的行都往上移一行
   if (ROLL_CLEAR_ITER == 0) begin
      // 清空的初始化操作
      if (flag_keys_write)
         keys_index_helper <= keys_index_helper - 70;
      cursor <= cursor - 70;
      y_cnt <= y_cnt - 1;
      roll_cnt <= roll_cnt - 70;
      roll_cnt_lines <= roll_cnt_lines - 1;
      roll_cnt_max <= roll_cnt_max - 70;
   end

   if (ROLL_CLEAR_ITER < 57) begin
      enter[ROLL_CLEAR_ITER] <= enter[ROLL_CLEAR_ITER + 1];
   end

   if (ROLL_CLEAR_ITER < 3990) begin    // 57 * 70
      ROLL_CLEAR_ITER <= ROLL_CLEAR_ITER + 1;
      keys[ROLL_CLEAR_ITER] <= keys[ROLL_CLEAR_ITER + 70];
   end else begin
      ROLL_CLEAR_FIRST_LINE <= 0;
      ROLL_CLEAR_ITER <= 0;
   end
end
```

图 3.26 滚屏清除第一行

在清除第一行前，一共有 n 行；我们的目的是清除第一行后，共有 n-1 行。所以，实际上是行数减少了 1，光标减少一行的长度，滚屏信息也需要减少一行；此外，也需要实现哪些行有命令提示符的滚动。ROLL_CLEAR_ITER 是一个计数变量，从 0 遍历到 3990。

自动滚屏

SSshell 规定，从 27 行起，行数再增加时显示的所有行都向上移动（即只修改滚屏相关的三个变量）；而从 57 行起，增加了这一行就需要清除第一行并腾出 4000 多个周期来处理滚屏问题。

```
if (y_cnt >= 27) begin                          // 27行后自动滚屏
   roll_cnt <= roll_cnt + 70;
   roll_cnt_lines <= roll_cnt_lines + 1;
   roll_cnt_max <= roll_cnt_max + 70;
end
```

图 3.27 27 行滚屏逻辑

上图展示了 27~56 行如何实现滚屏。

```
ROLL_CLEAR_FIRST_LINE <= (y_cnt >= 56);
```

图 3.28 57 行滚屏逻辑

上图展示了 57 行起如何实现第一行的清除，也很简单，让清除信号为 1 即可。

手动滚屏

SSshell 有手动滚屏功能：上下方向键可以自由滚屏，但加上了范围限制：不能向上滚出屏幕，当然也不能向下滚到没有那么多行的地方。

```
////////////////// Direction Key ////////////////////
if (direction_flag) begin                          // 方向键
    case (scanCode_E0)
        8'h75: begin    // 上
            if (roll_cnt_lines > 0) begin
                roll_cnt_lines <= roll_cnt_lines - 1;
                roll_cnt <= roll_cnt - 70;
            end
        end
        8'h72: begin    // 下
            if (roll_cnt_lines < roll_cnt_max) begin
                roll_cnt_lines <= roll_cnt_lines + 1;
                roll_cnt <= roll_cnt + 70;
            end
        end
```

图 3.29 手动滚屏逻辑

3.3.3 退格与回车逻辑分析

退格键

我们主要要考虑的问题是：在当前位置能不能退格?

目前仅有一种情况是不可以退格的：

- 这一行有命令提示符，而且这一行的光标位置恰在命令提示符后。

因为我们默认 SSshell 的初始状态就有命令提示符，所以如果整

个 keys 都没有字符可删，这一规则也仍然保证了退格键不奏效。

如果这一行没有命令提示符，说明这一行长度超过了一行而换到了下一行。这时，我们退格时回到上一行即可。

```
////////////////// Backspace //////////////////////
if (scanCode == 8'h66 && cursor > BASH_HEAD_LEN) begin// 退格键
   // keys[cursor - 1] <= 0;
   // 防止织毛衣，交给下个周期做
   flag_keys_write <= 1;
   keys_index_helper <= cursor - 1;
   keys_ASCII_help <= 0;

   // 处理x_cnt和y_cnt
   if (enter[y_cnt] && x_cnt == BASH_HEAD_LEN) begin  // 命令提示符这行到头了
      // Do nothing
   end else if (x_cnt == 0) begin
      // 回到上一行逻辑(这一行无命令提示符)
      // 一定有y_cnt > 0，因为第一行是有命令提示符
      x_cnt <= 69;
      y_cnt <= y_cnt - 1;
      cursor <= cursor - 1;
      if (roll_cnt_lines > 0) begin
         roll_cnt <= roll_cnt - 70;
         roll_cnt_lines <= roll_cnt_lines - 1;
         roll_cnt_max <= roll_cnt_max - 70;
      end
   end else begin                          // 普通退格逻辑
      x_cnt <= x_cnt - 1;
      cursor <= cursor - 1;
   end
end
```

图 3.30 退格键逻辑

这张图应该很好地展示了我们的思路。

回车键

当前阶段，回车时暂且认为我们将数据传递给外界模块（这是不是很像控制台的输入功能呢？），然后等待外部模块的处理。

回车的操作就更简单一点：只需要维护 y_cnt、cursor、滚屏信息等到正确值即可；记得动态维护 ROLL_CLEAR_FIRST_LINE。

```
//////////////// Enter ////////////////////
if (scanCode == 8'h5A || scanCode_E0 == 8'h5A) begin  // 回车键

   y_cnt <= y_cnt + 1;
   x_cnt <= 0;
   cursor <= cursor + (70 - x_cnt);
   ROLL_CLEAR_FIRST_LINE <= (y_cnt >= 56);

   if (y_cnt >= 27) begin                                  // 27行后自动滚屏
      roll_cnt <= roll_cnt + 70;
      roll_cnt_lines <= roll_cnt_lines + 1;
      roll_cnt_max <= roll_cnt_max + 70;
   end
end
```

图 3.31 回车键逻辑

在这里，我们给出一个流程图来总结一下。

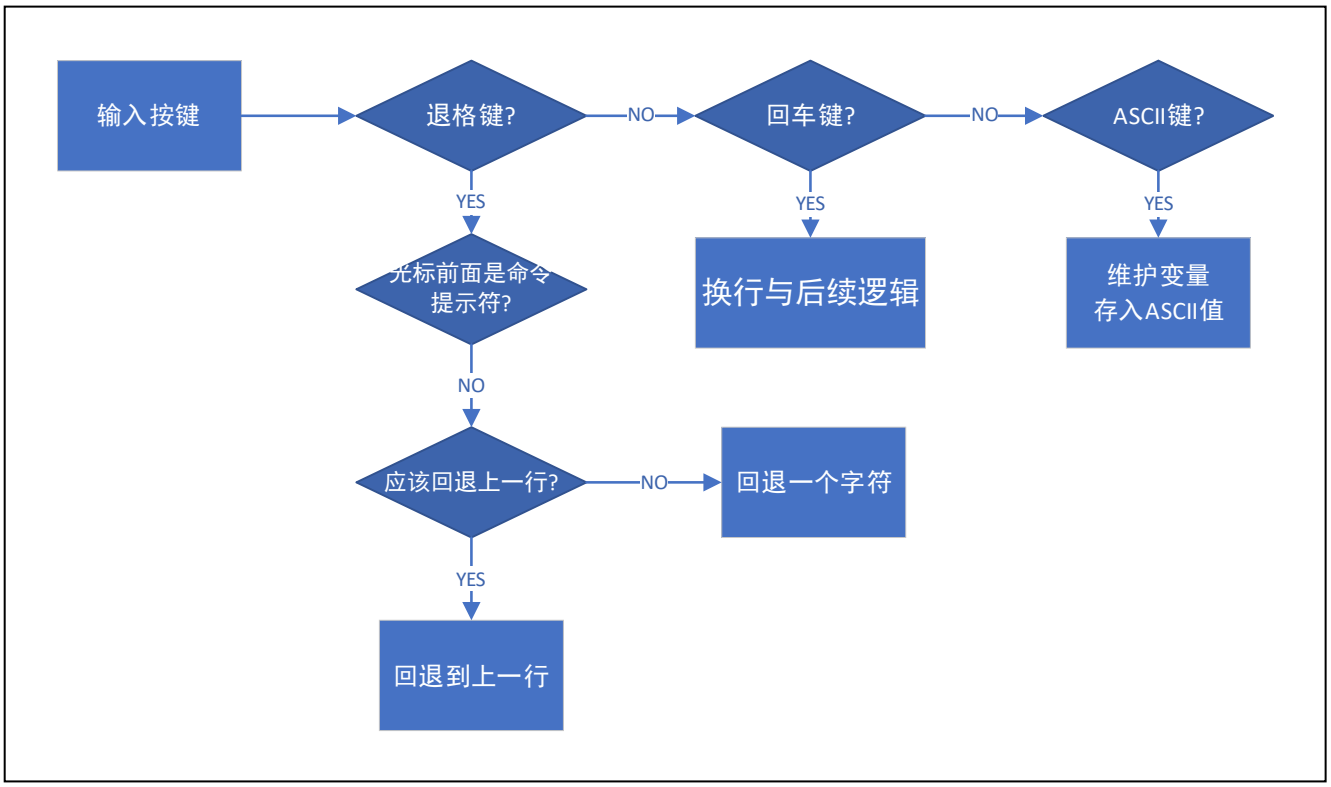


图 3.32 退格/回车流程图

3.3.4 多种配色方案

这是 SSshell 的创新功能：多种配色方案。我们挑选并实现了 4 种配色方案，效果如下（这里采用 HTML 网页的方式展示）

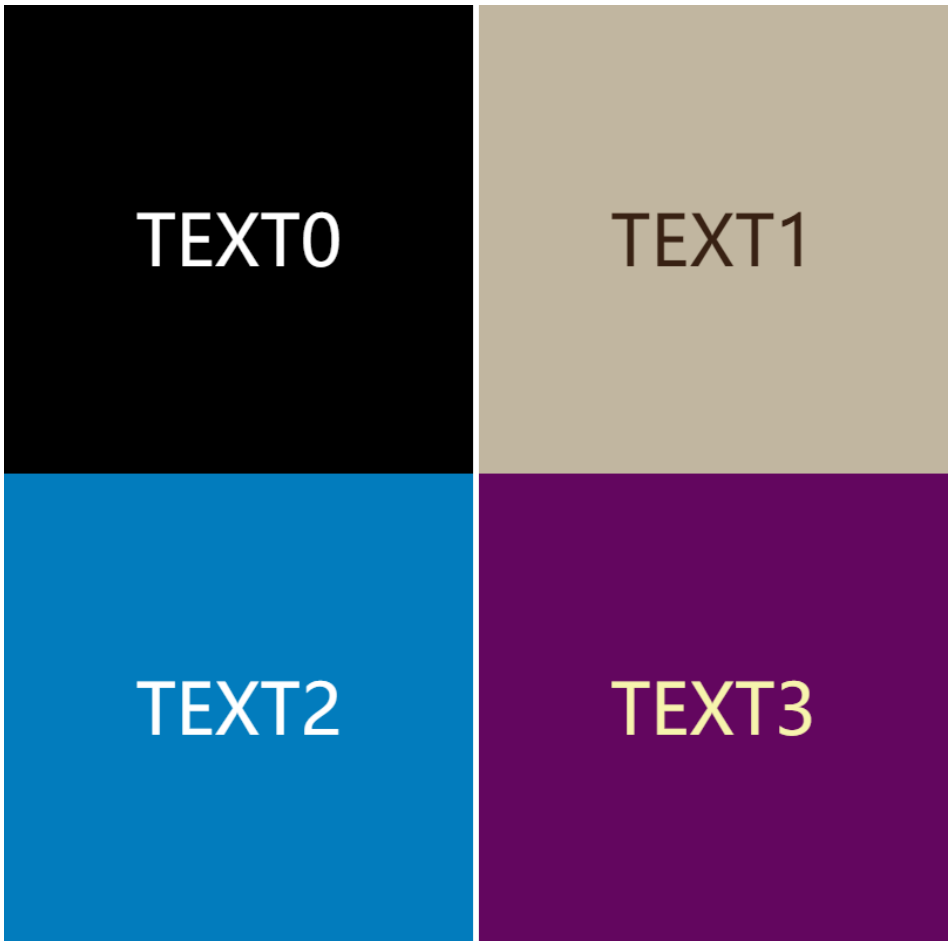


图 3.33 多种配色方案

该展示 HTML 文件也在 report 文件夹下，您可以自行查看效果。

用户使用时，只需要按下左右方向键，即可从正反两种顺序切换配色方案。

3.3.5 开机界面与动画

这是 SSshell 的创新功能：开机界面与动画。

这里，我们希望开机界面采用另外一个独特的模块 Welcome，而不依赖于 VideoMemory 复杂的逻辑。

先展示我们的开机界面：

图 3.34 开机界面

开机界面的存储

我们采用了基于实验 9 的图片存储方式，但！最终占用的存储空间只有 0.625M。我们设计的图片只有四种颜色：白色，绿色，黑色，米色。所以我们的图片实际上存储只用了 2 位（640×512），最终占用的空间很小（其实这幅图片的制作以及 mif 生成都及其复杂，这里就不再赘述了）。

注意，从顶层模块的角度看，它的主要功能和 VideoMemory 相似：读取(h_addr, v_addr)，并输出颜色值。

首先要根据坐标点拿到存储器内的数：

```
wire [18:0] address = {h_addr[9:0], v_addr[8:0]};

            welcomeStorage WStorage(
                .address(address),
                .clock(clk),
                .q(colorMode)
            );
```

图 3.35 开机界面存储器

然后把存储器里的数还原成对应的颜色：

```
assign rgb_welcome = (
    (real_v_addr >= 480) ?  // 包含过大值和负值情况
    24'h0 :
    getColorByMode(colorMode)
);

function [23:0] getColorByMode;
input [1:0] mode;
begin
    case (mode)
          0: getColorByMode = 24'h000000;  // 黑色
          1: getColorByMode = 24'hD2C4C1;  // 米色
          2: getColorByMode = 24'hFFFFFF;  // 白色
    default: getColorByMode = 24'h00C513;  // 绿色
    endcase
end
endfunction
```

图 3.36 开机界面颜色还原

实际屏幕分辨率是 640×480，纵坐标存 512 的目的是方便计算下标（只需要移位）。

开机界面进入退出动画

我们设计了开机界面从上面移动到中间，再从中间移动到下面直到退出屏幕的动画效果。我们采用了状态机的方式来实现动画。

首先，也就是状态 0，在最初的一段时间内什么也不做（因为 VGA 接收到有效信号到屏幕开始显示是有延迟的），经测试等待 4 秒后再开始动画在屏幕上观看的效果会比较好。

我们用 offsetY1 来处理从上面移动到中间的进入动画，offsetY2 来处理从中间移动到下面的退出动画。还是通过对 v_addr 进行一些加减计算来实现显示的移动效果。

```
                    // Initial
                    offsetY1 = 480;
                    offsetY2 = 0;
wire [12:0] real_v_addr = v_addr + offsetY1 - offsetY2;
wire [18:0] address = {h_addr[9:0], real_v_addr[8:0]};
```

图 3.37 开机界面动画输出逻辑

有了这些铺垫，最终如何实现两个 offsetY 的变量修改就变得非常明显了：

```
always @(posedge welcomeclk) begin
   if (state == 0) begin                    // StartTime
      if (count < StartTime) begin
         count <= count + 1;
      end else begin
         count <= 0;
         state <= 1;
      end
   end else if (state == 1) begin           // 从上面向中间滚动
      if (count < ScrollTime) begin
         count <= count + 1;
      end else begin
         count <= 0;
         if (count2 < 480) begin
            count2 <= count2 + 1;
            offsetY1 <= offsetY1 - 1;
         end else begin
            count2 <= 0;
            state <= 2;
            KeyPressAccess <= 1;
         end
      end
   end else if (state == 2) begin           // 在中间停留
      if (KeyPressed)
         state <= 3;
   end else if (state == 3) begin           // 从中间向下面移动
      if (count < ScrollTime) begin
         count <= count + 1;
      end else begin
         count <= 0;
         if (count2 < 480) begin
            count2 <= count2 + 1;
            offsetY2 <= offsetY2 + 1;
         end else begin
            count2 <= 0;
            state <= 7;                     // 空状态
         end
      end
   end else begin
      inWelcome <= 0;
   end
end
```

图 3.38 开机界面平移逻辑

state 最初为 0，为停滞状态；达到开始时间时（这里设定为 4 秒），开始从上面向中间滚动。KeyPressAccess 是是否允许探测按键的标志，仅有它为 1 的时候才去监测按键，如下。

```
always @(posedge newKey) begin
    if (KeyPressAccess)
        KeyPressed <= 1;
end
```

图 3.39 开机界面按键监测

在中间停留时（state=2），要实现“Press any key to continue”的功能。检测到按键时，向下移动。直到完全退出屏幕，inWelcome 输出为 0，这时向外传递一个信号：开机界面已经结束了。这可以用于顶层模块交互。

开机界面与 VideoMemory 的互斥配合

刚刚，我们看到了 inWelcome 这个变量。最初它被设置成 1，代表正在开机界面中。在这个时候，VideoMemory 不应该处理任何键盘消息（如果用户在开机界面按键，结果进入控制台时看见很多字母，会很奇怪，对吧）。

在顶层模块，对 VideoMemory 和 Welcome 两个模块输出的颜色再进行一次选择：

```
assign real_output_rgb = (inWelcome ? rgb_welcome : rgb);
```

图 3.40 Welcome 与 VideoMemory 的互斥输出

另外，在 VideoMemory 中接入 inWelcome 信号，如果它为 1 就不处理键盘消息即可。这个逻辑比较简单，就不展示了。

3.4 对外接口：Super I/O

这一节介绍 SSshell 显存的最后一部分功能：Super I/O。这是 SSshell 自己设计的 I/O 机制。我们力求显存独立于 CPU 存在，拦截键盘相关的操作；设计好显存后，只需要与 CPU 完成交互对接，即可把 CPU 的计算、执行、操作主存等功能与显示、键盘分隔开！

开机界面的对接非常容易，在 3.3.5 节中已经展示。所以本节只介绍如何与 VideoMemory 对接！这个对接机制比较复杂，为了便于读者理解，这里也分成了若干小节。

3.4.1 输入内容的存储

SSshell 规定，有效的一行输入不超过 128 字符。这是因为实际执行时，这么长的一行已经能够满足需求。当然，这也是 parameter 控制的，改起来也很方便。

```
                parameter BUFFER_LEN = 128;
  reg [12:0] out_lineLen_help;          // 向外输出长度的辅助变量
  reg [7:0] buffer  [BUFFER_LEN-1 : 0];

////////////////// Other ASCII Key ////////////////////
if (isASCIIkey) begin   // 其他正常字符键
   out_lineLen_help <= out_lineLen_help + 1;
   if (out_lineLen_help < BUFFER_LEN) begin  // 维护输出字符串
      buffer[out_lineLen_help] <= ASCII;
   end
```

图 3.41 输入内容的存储

图 3.41 展示了主要代码逻辑。其中，out_lineLen_help 相当于循环变量，一直是存入 buffer 的下标，同时也是这一行已经输入的长度；我们不断地存入输入的字符即可。当然，在退格时也要维护 out_lineLen_help 的值；由于维护较复杂，这里略去。

3.4.2 向外部模块传递用户输入行

存储了输入的内容，接下来要看看如何向外传递输入行。这里借用实验 8 的思路，引入 I/O 机制：

```
// 向外界模块输出bash输入信息，外部模块应该注意最后一位是00
input           lineOut_nextASCII,          // 外界模块读好一个字符之后应该传递1进来一个周期
output   reg    out_newASCII_ready,
output   reg    [12:0]   out_lineLen,       // 约定合法的一行最长BUFFER_LEN字符+00结束，值为实际长度
output          [7:0]    lineOut            // 输出，一个一个输出
```

图 3.42 向外界模块输出用户输入行-引脚

在 SSshell 中，用户回车后，显存将禁用键盘输入功能，准备向外部模块（也就是 CPU）传递这一行字符；SSshell 规定，显存向外界传递字符时，应该让 out_newASCII_ready 为 1，同时 lineOut 为输出的字符；显存在接收到 lineOut_nextASCII 的信号为 1 时，将输出下一个字符。输出所有有效字符后，还会输出一个'\0'，此时再读到 lineOut_nextASCII 为 1，则会在下一个周期关闭 out_newASCII_ready。

让我们细致地描述一下全过程：

- 用户敲回车后，VideoMemory 进入锁定状态，此时键盘输入将没有任何效果；同时，VideoMemory 会执行如下操作：
  - **将 out_newASCII_ready 置为 1**；
  - **在 lineOut 处输出这一行的第一个字符**（就算是空行，也有结尾符'\0'）；
  - **输出 out_lineLen 是这一行的有效长度**（0~128），不含结尾符'\0'。
- 外界模块对接操作：
  - 收到 out_newASCII_ready 信号后读取 lineOut 进行操作，然后**根据情况置 lineOut_nextASCII 为 1 索取下一个字符**；
  - 之后 VideoMemory 会向外输出下一个字符，直到输出结尾符'\0'后，

out_newASCII_ready 变为 0，输出结束。

- 外界检测输出结束的标志可以是'\0'，也可以是 out_newASCII_ready 变为 0，但是 ready 晚来一个周期。外界模块应该灵活使用。

这就是全过程了，但我们还需要注意几个细节：

- 回车后，可能会引起滚屏清空第一行！这时我们如何处理滚屏和输出的细节？

根据滚屏清空第一行优先的原则，我们不应该，或最好不要在检测到用户回车的周期开启输出模式！因为滚屏清空时不会进行别的操作，外界可能会读到错误的信息。我们把开启输出模式的操作缓冲一个周期，如图所示：

```
// 回车的下个周期：开始往外送数据，因为考虑到可能会引起57行之后的清空操作，所以隔一个周期处理
if (output_flag) begin
   output_flag <= 0;
   if (out_lineLen_help > BUFFER_LEN)
      out_lineLen <= BUFFER_LEN;
   else
      out_lineLen <= out_lineLen_help;
   out_lineLen_help <= 0;
   out_newASCII_ready <= 1;            // 空行也必须向外传递，否则无法完成处理
   keyboard_valid <= 0;
   running_program <= 1;
end
```

图 3.43 向外界模块输出用户输入行-缓冲

其中，running_program 是我们把用户输入行转交给 CPU 的标志。我们可以形象化地认为它代表程序（CPU）正在运行。

out_lineLen 是我们向外传递的这一行的有效长度。外界读取这个变量的值，就知道这一行有多长了（这个变量也可以省去）。

keyboard_valid 是是否允许接收键盘输入的一个控制变量。如果它为 1，键盘输入才会奏效。

这里展示如何实现整个输出流程：

```
// 输出总线
assign lineOut = (
    (out_lineLen_help == out_lineLen) ?
    0 :
    buffer[out_lineLen_help]
);

////////////////// Output lineOut Coding ////////////////////
// 屏幕输入，向外界输出逻辑
if (!keyboard_valid)                    // 键盘不能输入才执行，防止与顶层模块交互错误（保险机制）
   if (out_newASCII_ready) begin        // 数据输出逻辑
      if (out_lineLen_help == out_lineLen) begin
         out_newASCII_ready <= 0;
         out_lineLen_help <= 0;
      end else if (lineOut_nextASCII) begin
         out_lineLen_help <= out_lineLen_help + 1;
      end
   end
```

图 3.44 向外界模块输出用户输入行-输出

直到输出完全部的有效字符以及结束符'\0'，ready 才会变成 0。注意，我们只需要让 out_lineLen_help 不断加 1 即可完成输出的变化，因为 lineOut 是 assign 的。

3.4.3 从外部模块读入行并输出到 VGA

外界向显存传递内容并由显存输出到 VGA 和上一小节中显存向外界模块输出的逻辑差不多。

```
// 外界模块输入bash输出信息，外部模块应该注意最后一位是00
output   reg    lineIn_nextASCII,
input           in_newASCII_ready,          // 这一行的ready，这一行结束时应该为0
input           [7:0]    lineIn,            // 输入
```

图 3.45 从外界模块读入行并输出到屏幕-引脚

- 外界模块应该**模拟指令的执行过程**，将**输出**传给 VideoMemory：
  - **在 lineIn 接口处准备好要输出的第一个字符**（空行也应输出结束符 0）；
  - **让 in_newASCII_ready 为 1**，VideoMemory 将开始读入数据；
  - **检测到 lineIn_nextASCII 为 1 后，移动指针输出下一个字符**（直到输出结束符 0），因为 VideoMemory 读入完成后会让 lineIn_nextASCII 为 1 一个周期；
  - **如果这一行已经输出完了，但程序输出结果有多行，在输出这一行的结**

束符 0 后继续输出即可（VideoMemory 探测到结束符后会自动换行）；

SSshell 的输出比较灵活。我们不会要求 CPU 在执行程序后一口气把输出全部交给显存，而是希望在**运行时可以自由地向屏幕输出内容**；我们也不会希望 CPU 执行时只能输出一大串不含换行的内容，而是希望 **CPU 可以输出多行内容**。

所以，SSshell 约定：外界模块向 VideoMemory 传递的字符如果是'\0'，VideoMemory 会将其转换为换行。

```
375     ////////////////// Input lineIn Coding ////////////////////
376     // 外界输入，向屏幕输出逻辑
377     if (!keyboard_valid)                    // 键盘不能输入才执行，防止与顶层模块交互错误（保险机制）
378        if (lineIn_nextASCII) begin
379           lineIn_nextASCII <= 0;
380        end else begin
381           if (in_newASCII_ready) begin      // 有数据输入
382              lineIn_nextASCII <= 1;
383              if (lineIn == 0) begin         // 这行输出完了
384                 y_cnt <= y_cnt + 1;
385                 x_cnt <= 0;
386                 cursor <= cursor + (70 - x_cnt);
387                 ROLL_CLEAR_FIRST_LINE <= (y_cnt >= 56);
388
389                 if (y_cnt >= 27) begin                          // 27行后自动滚屏
390                    roll_cnt <= roll_cnt + 70;
391                    roll_cnt_lines <= roll_cnt_lines + 1;
392                    roll_cnt_max <= roll_cnt_max + 70;
393                 end
394
395              end else begin
396                 // 后续输出到屏幕逻辑
397                 //keys[cursor] <= lineIn;
398                 flag_keys_write <= 1;
399                 keys_index_helper <= cursor;
400                 keys_ASCII_help <= lineIn;
401
402                 cursor <= cursor + 1;
403                 // 处理x_cnt和y_cnt
404                 if (x_cnt == 69) begin
405                    y_cnt <= y_cnt + 1;
406                    x_cnt <= 0;
407                    ROLL_CLEAR_FIRST_LINE <= (y_cnt >= 56);
408
409                    if (y_cnt >= 27) begin                       // 27行后自动滚屏
410                       roll_cnt <= roll_cnt + 70;
411                       roll_cnt_lines <= roll_cnt_lines + 1;
412                       roll_cnt_max <= roll_cnt_max + 70;
413                    end
414                 end else begin
415                    x_cnt <= x_cnt + 1;
416                 end
```

图 3.46 从外界模块读入行并输出到屏幕-输出

输出逻辑也比较简单，只需要模拟一个字符从键盘被输入即可。但要注意的是，这里仍然可能需要滚屏；我们要注意，滚屏清空第一行时不会处理别的事情，这可能导致 lineIn_nextASCII 一直为 1，导致后续操作的错误！所以要记得在滚屏清空第一行时把它还原为 0。

```
////////////////// Screen Rolling Coding ////////////////////
if (ROLL_CLEAR_FIRST_LINE) begin        // 滚屏到57行了，把后面的行都往上移一行
   if (lineIn_nextASCII)
      lineIn_nextASCII <= 0;
   if (ROLL_CLEAR_ITER == 0) begin
      // 清空第一行的初始化操作
      if (flag_keys_write)
         keys_index_helper <= keys_index_helper - 70;
      cursor <= cursor - 70;
      y_cnt <= y_cnt - 1;
      roll_cnt <= roll_cnt - 70;
```

图 3.47 从外界模块读入行并输出到屏幕-细节处理

3.4.4 程序执行的 I/O 控制流程

这一小节，我们介绍 SSshell 约定的显存视角下的程序运行是怎么样的。

首先，用户在键盘上输入一行内容并回车，SSshell 把它传递给 CPU（CPU 将去调用命令解析功能并执行响应的程序）；SSshell 约定，显存只能处在等待用户输入，或者接受 CPU 输出的状态之一（如果一个控制台可以同时输入/输出，是不是有些流氓呢？）

但我们在 3.4.3 中约定，CPU 传递的'\0'被认为是换行。我们也不能简单以 CPU 传递进来的 in_newASCII_ready 变为 0 来作为 CPU 处理的结束标记（记得吗？在 3.4.3 中我们希望 CPU 可以随意输出）。怎么办呢？我们必须增加两个引脚：in_solved，out_solved。

```
input           in_solved,  // 结束信号，解决完这条指令后传递1一个周期进这个模块
output   reg    out_solved, // 本模块处理完结束信号会输出1一个周期
```

图 3.48 程序运行结束信号引脚

CPU 执行完这个程序后，只需要将 in_solved 拉高，并等待 VideoMemory 把 out_solved 拉高后把 in_solved 还原为 0 即可。

之所以这样设计，是因为 VideoMemory 可能正在滚屏清除第一行的过程。所以需要一个 out_solved 告诉外界拉低，外界再拉低。

而 VideoMemory 接受到 in_solved 之后的操作也比较简单：在行首添加命令提示符，改变光标位置和 x_cnt，并恢复键盘的功能，回到接受用户输入模式即可。

```
////////////////// Finish Output lineOut Coding ////////////////////
if (out_solved)
    out_solved <= 0;
else if (!keyboard_valid && in_solved) begin
    // 解决这条指令，恢复输入模式
    keyboard_valid <= 1;
    x_cnt <= BASH_HEAD_LEN;
    cursor <= cursor + BASH_HEAD_LEN;
    out_solved <= 1;
    enter[y_cnt] <= 1;    // 新的命令提示符
    running_program <= 0;
end
```

图 3.49 in_solved 处理

3.4.5 程序运行时索要用户输入

这是 SSshell 显存的特色功能。想象一下，可以输出一段字符串，然后再索要用户输入，是不是很令人激动？

在 SSshell 中，设计了这样的功能。但按照之前的设定，用户回车后键盘即锁定无效，进入专注接收 CPU 字符串并向屏幕输出的状态。现在，CPU 需要用户继续输入并回车，我们需要引入新引脚，做和 in_solved 和 out_solved 类似的操作：

```
input          in_require_line,  // 需要输入一行数据
output   reg   out_require_line, // 知道了，然后我开始输入（配合滚屏）
```

图 3.50 运行时索要用户输入-引脚

现在我们考虑一个问题：此前，我们约定退格键不生效的唯一条件是光标恰在命令提示符后。但，现在可以运行时输入，按理来说退格键在退到输出的内容的时候不应该再退了！此前的设计明显过时了，因为退格键可以删掉输出内容，甚至还会导致其他一系列未知错误。

我们来看一下 VideoMemory 是如何处理 in_require_line 信号的：

```
////////////////// Require new line ////////////////////////
if (!in_newASCII_ready && in_require_line) begin
   keyboard_valid <= 1;
   out_require_line <= 1;
   set_running_start_cursor <= 1;
end
if (set_running_start_cursor) begin // 下一个周期再去读cursor，这样可以防止各种意外
   set_running_start_cursor <= 0;
   running_start_cursor <= cursor;
end

if (out_require_line)
   out_require_line <= 0;
```

图 3.51 运行时索要用户输入-实现

图 3.51 中，running_start_cursor 是运行时索要输入时，光标能退格的极限位置。如果光标到这个位置，退格键就不能继续退！这样就解决了退格键删除 CPU 输出的尴尬局面。而且，一行内同时出现输出和输入变得可能了！set_running_start_cursor 则是一个标记变量，起缓冲作用，我们在下个周期再去设置 running_start_cursor。这也是一种保险机制，因为这个时钟周期可能对 cursor 还有修改，但下个周期应该不会。

```
////////////////// Backspace //////////////////////
if (scanCode == 8'h66 && cursor > BASH_HEAD_LEN) begin// 退格键
   // keys[cursor - 1] <= 0;
   // 防止织毛衣，交给下个周期做
   flag_keys_write <= 1;
   keys_index_helper <= cursor - 1;
   keys_ASCII_help <= 0;

   // 处理x_cnt和y_cnt
   if (enter[y_cnt] && x_cnt == BASH_HEAD_LEN) begin  // 命令提示符这行到头了
      // Do nothing
      out_lineLen_help <= 0;
   end else if (x_cnt == 0 && (
         (!running_program) || (cursor > running_start_cursor)
         )) begin
      // 回到上一行逻辑(这一行无命令提示符)
      // 要么是没运行程序，要么是程序需要输入
      // 如果程序需要输入，不能在需要输入的地方顶头退格！会把上一行退掉的。
      // 一定有y_cnt > 0，因为第一行是有命令提示符的
      out_lineLen_help <= out_lineLen_help - 1;
      x_cnt <= 69;
      y_cnt <= y_cnt - 1;
      cursor <= cursor - 1;
      if (roll_cnt_lines > 0) begin
         roll_cnt <= roll_cnt - 70;
         roll_cnt_lines <= roll_cnt_lines - 1;
         roll_cnt_max <= roll_cnt_max - 70;
      end
   end else if (
      ((!running_program) && (x_cnt > 0)) ||
      ((running_program) && (cursor > running_start_cursor))
   ) begin                          // 普通退格逻辑
      out_lineLen_help <= out_lineLen_help - 1;
      x_cnt <= x_cnt - 1;
      cursor <= cursor - 1;
   end
end
```

图 3.52 运行时索要用户输入-退格

图 3.52 中，**红框内部分**处理退格键退到头（这一行没有命令提示符）的情况：要么当前不在运行程序，要么运行程序索要输出，并且退格仍然合法（不会删除输出的内容）。我们需要维护滚屏信息、x_cnt、y_cnt、cursor 等。**绿框内部分**处理普通退格逻辑，逻辑也比较明显。

至此，SSshell 的显存 VideoMemory 已经完成其全部功能！我们也有一个与其对接的示例外部模块 EchoExample 随实验文件给出，测试结果完美。EchoExample 的功能是连续回显三次用户输入的内容。

VideoMemory 的逻辑过于复杂，但需要说明的地方在本实验报告中已经全部叙述完毕。还有更多复杂的细节这里就不赘述了。