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

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。

目　录

Catalog

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:

✧ 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

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。

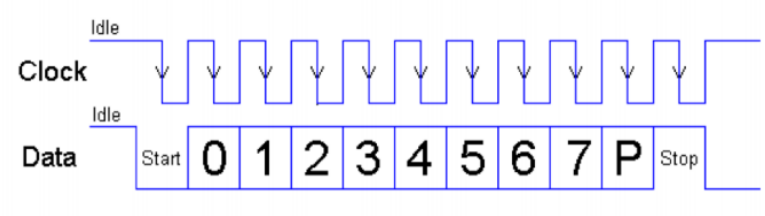

图 2.1 PS2 信号输出数据时序图

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。

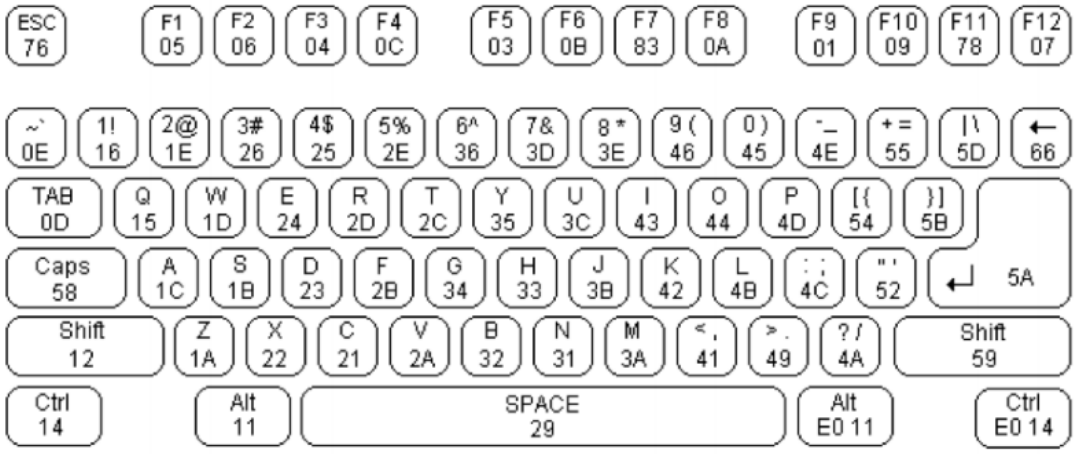

图 2.2 键盘扫描码 (主键盘)

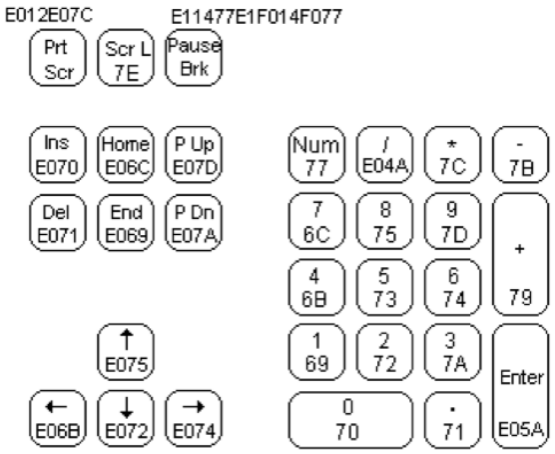

图 2.3 键盘扫描码 (扩展键盘与数字键盘)

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。

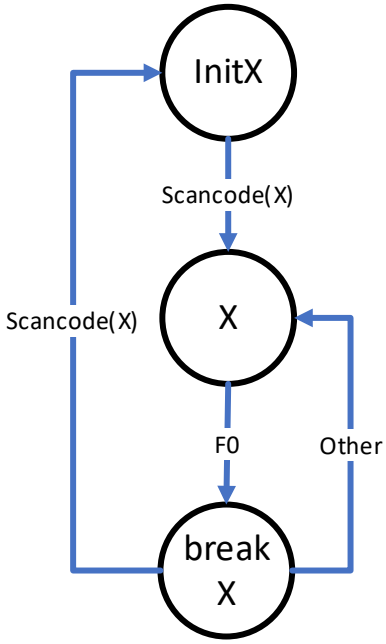

图 2.4 单个键对应的状态机

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。

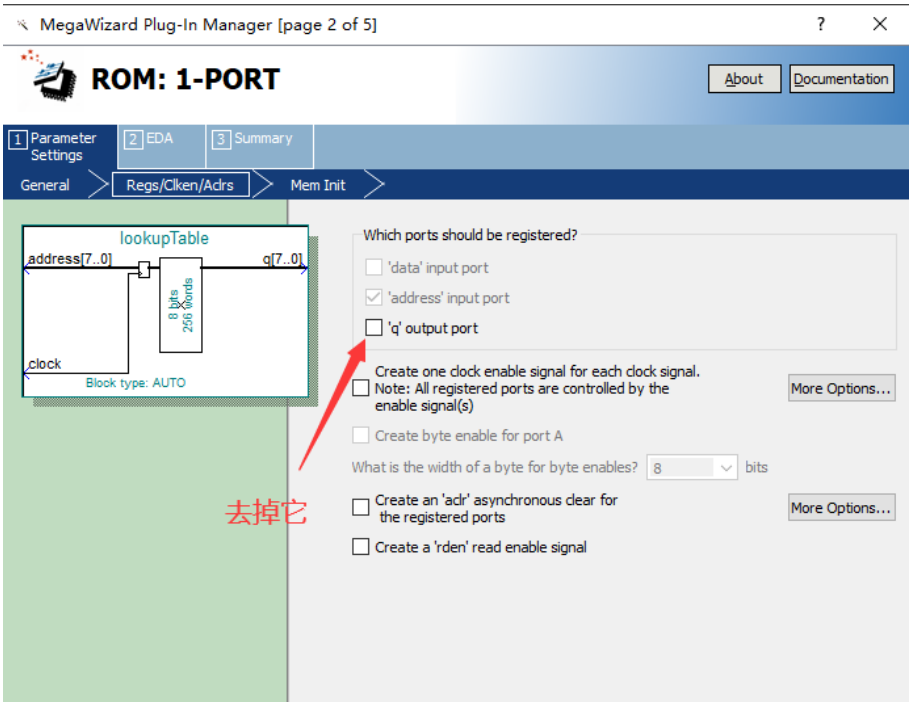

图 2.8 取消输出 q 的缓冲

```
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。

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