现代电子系统设计

FPGA 综合实验

手写数字识别

学　　号　　2017011589

姓　　名　　吾尔开希

专　　业　　自动化

日　　期　　2019.7.1---7.12

目录

1. 实验内容

用户在 LTM 屏上手写数字，可点击屏幕右 1/3 部分清空屏幕重新写入，写好后点击屏幕左 1/3 部分开始识别。LCD 屏用于显示提示信息与识别结果，在识别过程中，由于卷积网络计算时间较长，LCD 屏显示 0 到 100%的识别进程，识别结束后 LCD 屏显示结果，用户可进行下一次写入与识别。另外，数码管实时显示触屏坐标。

2. 设计方案

2.1 LTM 屏

LTM 屏需要实现两个功能：读取接触点坐标，显示笔迹。

（1）读取接触点坐标

读取接触点坐标相对比较好实现，在 adc_spi_controller（取自示例工程）基础上，增加总线接口，并将 100MHz 的总线时钟二分频为 50MHz，得到 ltm_adc 模块。该模块连接 LTM 外设的 LTM_ADC_DIN, LTM_ADC_DOUT, LTM_ADC_BUSY 与 LTM_ADC_PENIRQ_n 信号，通过模拟量转数字量操作，得到 LTM 触摸点的两个 12 位坐标值。坐标值连接到数码管上，实时显示触摸点坐标。

```
//触屏识别模块
//from 示例工程DE2_115_LTM_EPHOTO
//加了总线接口
module adc_spi_controller (
        //---------------------- Avalon -----------------------
        csi_clk,                //100MHz Clock
        csi_reset_n,
        avs_chipselect,
        avs_address,
        avs_read,
        avs_readdata,
        avs_write,
        avs_writedata,
        //---------------------- User Interface --------------------
        coe_iADC_DOUT,
        coe_iADC_BUSY,
        coe_iADC_PENIRQ_n,
        coe_oTOUCH_IRQ,
        coe_oX_COORD,
        coe_oY_COORD,
        coe_iRST_n,
        coe_oADC_DIN,
        coe_oADC_DCLK,
        coe_oADC_CS,
);

// Avalon
input                       csi_clk;
input                       csi_reset_n;
input                       avs_chipselect;
input       [3:0]       avs_address;
input                       avs_read;
output reg  [31:0]      avs_readdata;
input                       avs_write;
input       [31:0]      avs_writedata;
// User Interface
output                      coe_oADC_DCLK;
output                      coe_oADC_CS;
output                      coe_oTOUCH_IRQ;
output      [11:0]          coe_oX_COORD;
output      [11:0]          coe_oY_COORD;
output                      coe_oADC_DIN;
```

总线连接：

```
.ltm_adc_0_conduit_end_0_export_irstn           (DLY0),           //
.ltm_adc_0_conduit_end_0_export_oadc_din        (LTM_ADC_DIN),
.ltm_adc_0_conduit_end_0_export_oadc_dclk       (adc_dclk),       //
.ltm_adc_0_conduit_end_0_export_oadc_cs         (),         //
.ltm_adc_0_conduit_end_0_export_iadc_dout       (LTM_ADC_DOUT),
.ltm_adc_0_conduit_end_0_export_iadc_busy       (LTM_ADC_BUSY),
.ltm_adc_0_conduit_end_0_export_iadc_penirqn    (LTM_ADC_PENIRQ_n),
.ltm_adc_0_conduit_end_0_export_otouch_irq      (touch_irq),       //
.ltm_adc_0_conduit_end_0_export_o_xcoord        (x_coord),        //
.ltm_adc_0_conduit_end_0_export_oy_coord        (y_coord),        //
```

（2）显示笔迹

显示笔迹较难实现，由于 LTM 屏的显示需要一个单独的模块与之配合，在同样的时钟频率下根据 LTM 屏的读使能信号，逐个给出每个像素点的 RGB 值，即隔一段时间进行一次扫描显示。与之配合的模块需要有存储能力，另外需要挂接在总线上。由于 SDRAM 已经被 NIOS 核用作内存，所以不能使用，我选择使用 SRAM 作为存储外设，LTM 屏有 480*800 个像素点，每个像素点仅需存储一

字节的灰度值，所以 2MB 容量的 SRAM 是足够的。再设计一个专门的模块 mySRAM_controller，其作用是：读取 NIOS 核的输入，将每个像素点的灰度值存储在 SRAM 中；与 LTM 的控制模块交互，在其需要时传输每个像素点的灰度值；

LTM 屏的驱动模块有：Reset_Delay，产生特定的延时信号；lcd_spi_controller，三线控制；lcd_timing_controller，与 mySRAM_controller 交互，驱动 LTM 屏扫描显示。它们的实例化代码如下：

```
Reset_Delay          u8    (.iCLK(CLOCK_50),
                     .iRST(KEY[0]),
                     .oRST_0(DLY0),
                     .oRST_1(DLY1),
                     .oRST_2(DLY2)
                     );

lcd_spi_controller    u1     (
                     // Host Side
                     .iCLK(CLOCK_50),
                     .iRST_n(DLY0),
                     // 3 wire Side
                     .o3WIRE_SCLK(ltm_sclk),
                     .io3WIRE_SDAT(LTM_SDA),
                     .o3WIRE_SCEN(LTM_SCEN),
                     .o3WIRE_BUSY_n(ltm_3wirebusy_n)
                     );

lcd_timing_controller   u6  (
                     .iCLK(LTM_NCLK),
                     .iRST_n(DLY2),
                     // sdram side
                     .iREAD_DATA(ltm_SRAM_Read_DATA),
                     //.iREAD_DATA2(Read_DATA2),
                     .oREAD_SDRAM_EN(mRead),
                     // lcd side
                     .oLCD_R(LTM_R),
                     .oLCD_G(LTM_G),
                     .oLCD_B(LTM_B),
                     .oHD(LTM_HD),
                     .oVD(LTM_VD),
                     .oDEN(LTM_DEN)
                     );
```

mySRAM_controller 的接口：

```
//SRAM控制模块
//用于存储NIOS II 写入的数据，且向LTM扫描输出数据
module mySRAM_controller (
        //---------------------- Avalon -----------------------
        csi_clk,                //100MHz Clock
        csi_reset_n,
        avs_chipselect,
        avs_address,
        avs_read,
        avs_readdata,
        avs_write,
        avs_writedata,
        //---------------------- User Interface ---------------------
        //与SRAM硬件的接口
        coe_oSRAM_ADDR,
        coe_oSRAM_CE_N,
        coe_ioSRAM_DQ,
        coe_oSRAM_LB_N,
        coe_oSRAM_OE_N,
        coe_oSRAM_UB_N,
        coe_oSRAM_WE_N,
        //向LTM lcd_timing_controller输出信号
        coe_iLTM_clk,
        coe_iLTM_clk_irstn,
        coe_iLTM_mRead,
        coe_oLTM_Data,
);
```

mySRAM_controller 的硬件连接，有两个模式，一个是 NIOS 软核写，另一个是向 LTM 屏显示输出，不同的模式下，各种信号的连接不同：

```
reg IO_state;       //为1时写，为0时读

reg [19:0] write_address;
reg [19:0] read_address;
reg [7:0] write_data;
assign coe_oSRAM_CE_N = 0;          //片选始终有效
assign coe_oSRAM_LB_N = 0;      //只使用低位数据
assign coe_oSRAM_UB_N = 0;
assign coe_oSRAM_ADDR = (IO_state) ? write_address:read_address;
assign coe_ioSRAM_DQ = (IO_state) ? {8'h00,write_data}:16'hzzzz;
//向LTM扫描模块输出的数据，输出的RGB值相等，当写入数据时，输出0xFF
assign coe_oLTM_Data = (IO_state) ? 32'hFFFFFF00 : {coe_ioSRAM_DQ[7:0], coe_ioSRAM_DQ[7:0],
coe_ioSRAM_DQ[7:0], 8'h00};

assign coe_oSRAM_WE_N = ~IO_state;
assign coe_oSRAM_OE_N = IO_state;
```

mySRAM_controller 的写入控制与读写控制用状态机实现，由于写入数据容易丢失，所以设置两个状态，保证数据正确写入。NIOS 软核写入的 32 位数据中，20 位地址数据在第 27~8 位，灰度值在低 8 位：

```
reg [1:0] write_state;
parameter S0 = 0, S1 = 1, S2 = 2; //S0是读状态，S1是写状态1，S2是写状态2，用于延时

always @ (write_state)
begin
    case (write_state)
        S0:
        begin
            write_data <= write_data;
            write_address <= write_address;
            IO_state <= 0;
        end
        S1:
        begin
            write_data <= avs_writedata[7:0];
            write_address <= avs_writedata[27:8];//20位地址数据在第27~8位
            IO_state <= 1;
        end
        S2:
        begin
            write_data <= write_data;
            write_address <= write_address;
            IO_state <= 1;
        end
        default:
        begin
            write_data <= write_data;
            write_address <= write_address;
            IO_state <= 0;
        end
    endcase
end


always@(posedge csi_clk or negedge csi_reset_n)
begin
    if(!csi_reset_n)
    begin
        write_state <= S0;
    end
    else
    begin
        case (write_state)
                S0:
                begin
                    if((avs_chipselect == 1) && (avs_write == 1))
                        write_state <= S1;
                    else
                        write_state <= S0;
                end
                S1:
                    write_state <= S2;
                S2:
                    write_state <= S0;
                default:
                    write_state <= S0;
            endcase
    end
end
```

mySRAM_controller 向 LTM 屏的输出控制实现，每输出一个像素点的灰度值后，地址加 1：

```
always@(posedge coe_iLTM_clk or negedge coe_iLTM_clk_irstn)
begin
    if(!coe_iLTM_clk_irstn)
        read_address <= 0;
    else
    begin
        if(read_address >= 384000)
            read_address = 0;

        if(coe_iLTM_mRead == 1)
            read_address <= read_address + 1;
        else
            read_address <= read_address;
    end
end
endmodule
```

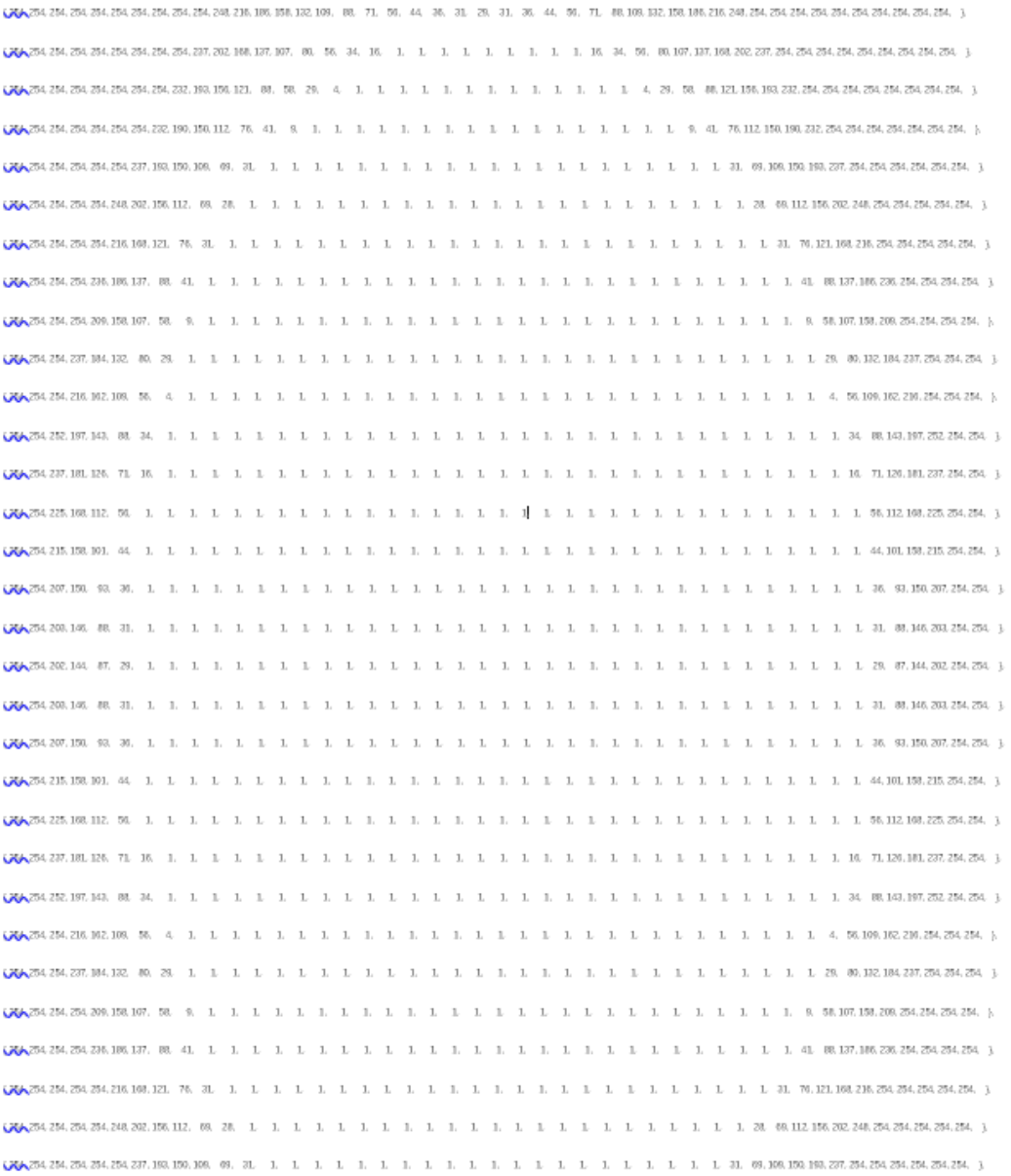
2.2 NIOS 软核逻辑

（1）收集接触点信息

NIOS 软核从 LTM_adc 模块读取接触点的坐标，先将其转化成标准坐标。

```
//将原始的坐标数据转化成标准坐标数据, 0<i<800, 0<j<480
void CoordTrans(unsigned int raw_x, unsigned int raw_y, unsigned int* i, unsigned int* j)
{
    *i = 800 - raw_y*800/4095;
    *j = 480 - raw_x*480/4095;
}
```

再将 480*480 大小的存储矩阵中对应位置为白色，然而，若只对一个像素点进行操作有两方面的不足，一方面，显示在 LTM 屏上的笔迹很细，不真实；另一方面，MNIST 数据库中笔迹较粗，而很细的笔迹与之不匹配，识别准确率将会大大降低。所以，我用 python 生成了灰度值矩阵，每次以接触点为中心，存储矩阵中周围 45*45 大小的区域都会按照这个灰度值矩阵的值来进行更新，起到接触点中间较黑，周围较白的效果。



（2）数据预处理

收集完接触点信息开始识别前，还要先降采样，将 480*480 的数据缩小为 28*28 的矩阵，再转置，标准化。

```
1.  //降采样将 480*480 的数据转化成 28*28
2.
3.  void shrinkData()
4.  {
5.      unsigned int small_i, small_j, i, j;
6.      double k = 28.0/480.0;
7.
8.
9.      for(i=0;i<28;i++)
```

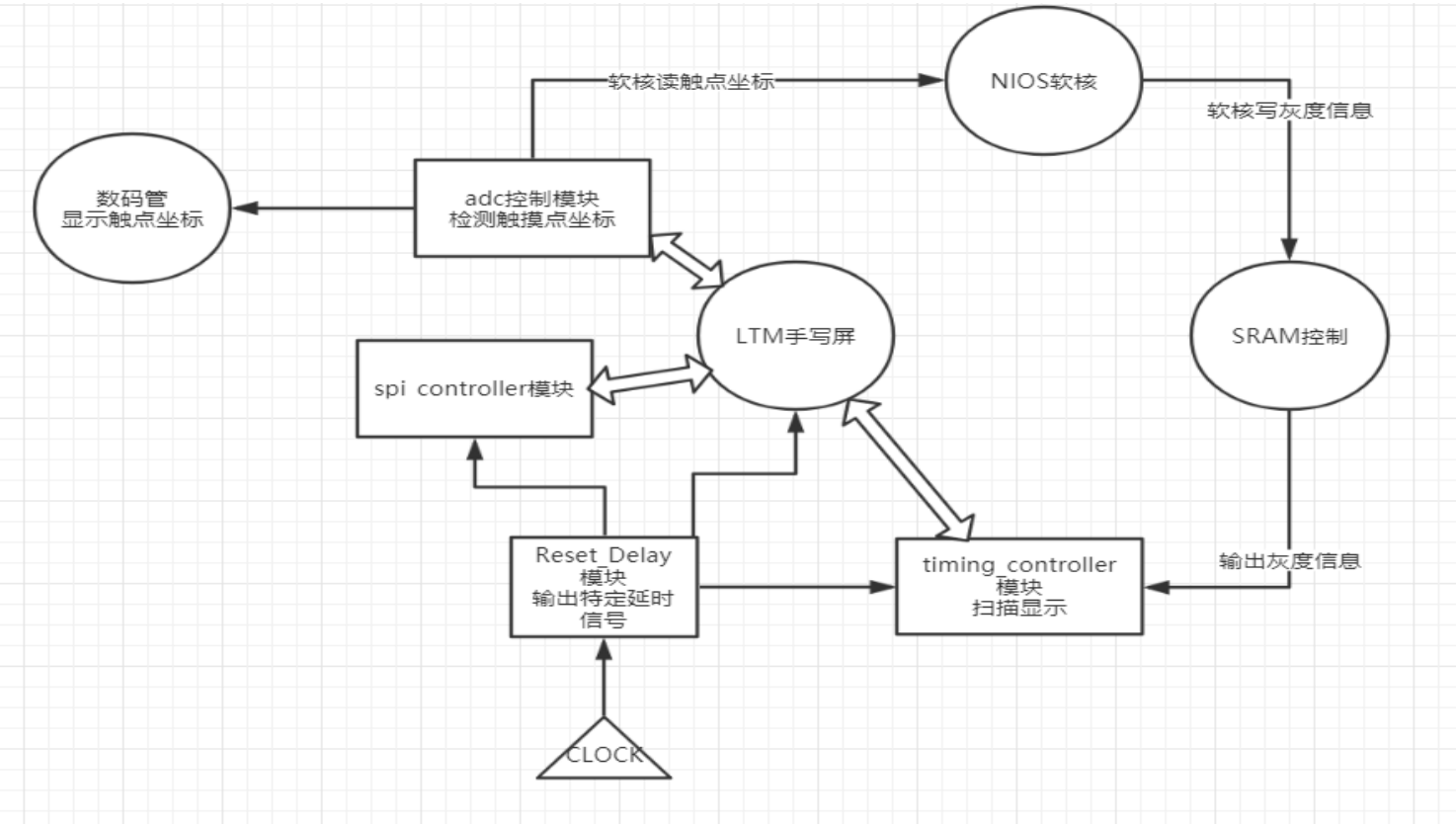
```
    {
        for(j=0;j<28;j++)
        {
            smallData[i][j]=0;
            smallDataCount[i][j]=0;
        }
    }

    for(i=0;i<480;i++)
    {
        for(j=0;j<480;j++)
        {
            small_i = (int)((double)(i)*k);
            small_j = (int)((double)(j)*k);

            smallData[small_i][small_j] += drawData[i][j];
            smallDataCount[small_i][small_j] += 1;
        }
    }

    for(i=0;i<28;i++)
    {
        for(j=0;j<28;j++)
        {
            std_data[i][j] = (double)smallData[j][27-
i]/(double)smallDataCount[j][27-i];
            std_data[i][j] = 1 - std_data[i][j]/255.0;
        }
    }
}
```

（3）神经网络运算

使用 CNN 先在电脑端用 MNIST 数据库进行训练，测试集上得到的准确率为 92%。用 c 语言复现卷积、maxpooling 等运算，利用训练好的模型参数在 NIOS 软核进行计算，得到识别结果。

```
self.cnn1 = nn.Conv2d(in_channels = 1, out_channels = 64, kernel_size = 3, stride = 1, padding = 0)
self.max1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
self.cnn2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=1, padding=0)
self.max2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
self.linear = nn.Linear(2304, 10)
```

3. 电路图



模块示意图如图所示，总体分为三大主要部分：LTM 手写屏、NIOS 软核和 SRAM 控制，一个辅助部分：数码管显示触电坐标。其中，LTM 手写屏由四个模块支撑，adc 控制模块检测触摸点坐标提供给 NIOS 软核读取。NIOS 软核向 SRAM 控制模块写入 SRAM 控制写入每个像素的灰度信息并存储在 SRAM 中，另外，SRAM 控制模块向 LTM 屏的 timing_controller 模块配合输出灰度信息，实现扫描显示。

4. 模块选择与参数设置

4.1 Reset_Delay 模块

给 LTM 其他模块提供特定的延时信号。

4.2 timing_controller 模块

驱动 LTM 屏的扫描显示。

4.3 adc 控制模块

检测触摸点坐标并提供给 NIOS 软核读取。

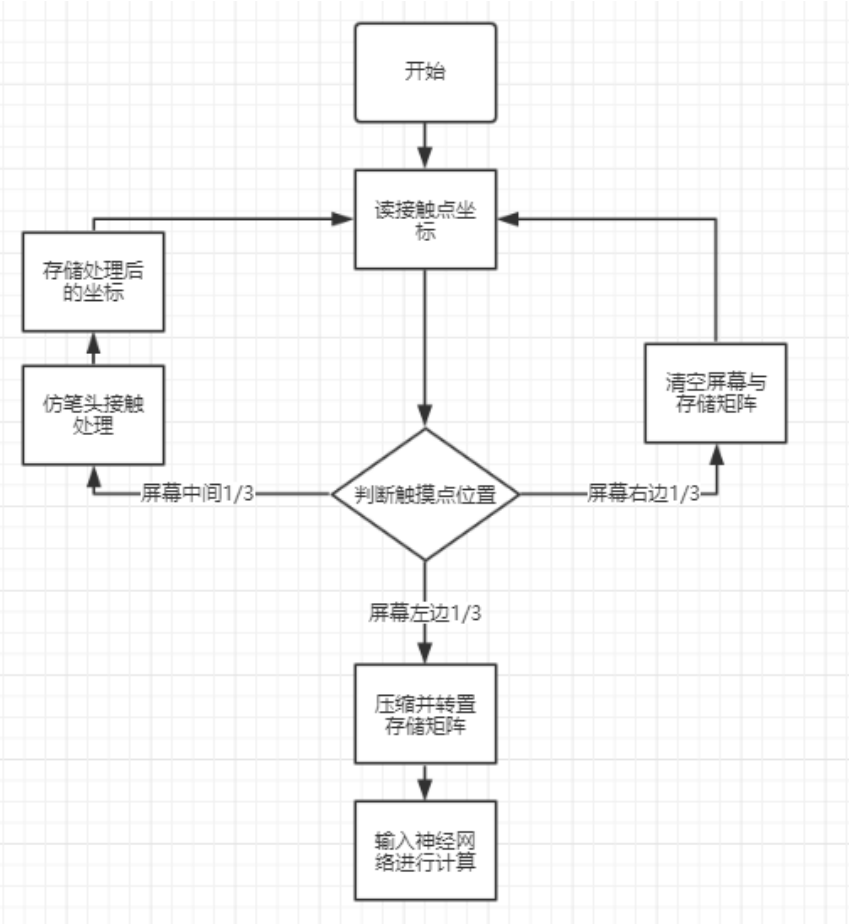
4.4 mySRAM_controller 模块

该模块一方面读取 NIOS 核的输入，将每个像素点的灰度值存储在 SRAM 中；另一方面与 LTM 的控制模块交互，在其需要时传输每个像素点的灰度值；

4.5 数码管模块

实时显示 LTM 屏接触点坐标。

5. 流程图



NIOS 软核中的逻辑控制流程图如图所示，图中只给出一次识别的流程，实际上是循环进行的。

6. 实验结果

达到了实验内容的要求，最初神经网络只有一层 fc，运算速度比较快，然而改成卷积网络后，虽然准确率提高了，但运算时间大大提高，所以增加了在 LCD 屏显示运算进程的功能。

在 0 到 9 这十个数字中，0 到 8 都能一次准确识别，9 会被识别成 8，与训练时 92%的准确率相近。

7. 实验中遇到的问题与解决方法

7.1 LTM 屏不显示

LTM 屏显示模块搭建连接好后，LTM 屏并不能正常显示，我起初以后是其他模块传入数据的问题，于是将传入的数据设为固定值，但还是不显示。后来我发现原来是 LTM 的一根 reset 引脚忘了连高电平，于是默认低电平，LTM 屏一直处于 reset 状态，自然不会显示。

7.2 卷积神经网络计算错误

用 c 语言部署好卷积神经网络后，我发现识别结果反而变差，于是我针对同一个输入，对比了 c 语言版本的输出与 python 版本的输出，发现二者不一样，说明 c 语言版本的实现有错误，于是我逐一对比了每一层的输出，发现最后一层出错，原来是维度错误，虽然不会数组越界，但是计算顺序错了，结果自然也是

错误的。改正这个错误后，计算结果的准确率恢复正常。

7.3 Verilog 基本语言错误

Verilog 定义 wire 或 reg 时，长度要写在变量名前（wire [31:0] ouput）而不是后面（wire ouput [31:0]）。

7.4 存储空间大小的问题

由于神经网络的参数较多，在写 NIOS 软核程序时，我发现在全局变量中加入其参数后，工程建立会报错。我认为这是存储空间大小的问题，求助了助教后，他告诉我全局变量和局部变量的存储位置不一样，于是我将参数放到函数里，使其变成局部变量，问题迎刃而解。

8. 体会、收获与建议

通过这次实验，独立搭建一个完整的系统，即锻炼了我的耐心，也提高了我解决问题、综合考量的能力。搭建硬件时遇到的问题比较多，我也用了很多方法来调试，显示屏最终正常运行时我感到十足的成就感。

用 c 语言复现 cnn 网络时出现的问题也花费了我比较长的时间来调试，如果当时更认真地写代码，我想这个问题也不会出现，这是我学到的教训，凡事不能急于求成，一定要认真走好每一步。

这个实验给了我很大的锻炼与成就感，感谢助教老师的帮助。