实验二：在 Zynq FPGA 上实现 LeNet 网络
实验报告

姓名：张安澜
学号：15061075
2018 年 1 月 2 日

一、摘要

在这个实验中，我在 Znqy FPGA 上实现了一个小型网络 LeNet 的前向传播过程，进行图片识别，并通过 10 张测试图片测试了实现效果，测试结果并不让人满意，最后我分析了可能导致这个结果产生的一系列原因进行总结。

二、实验目的

1、使用算法开发框架(Cafffe/TF/PyTorch/etc.)进行训练、测试,理解 LeNet 网络结构,熟知前向传播的计算过程,并将训练好的模型数据整理出来,待硬件实现用。

2、掌握使用 Verilog 语言设计卷积神经网络计算功能模块的方法,包括 CONV、ReLU、MaxPool、FC 层等。

3、掌握前向传播计算体系结构的设计方法,包括基本数据通路设计和实现、状态机设计和实现等。

4、掌握 AXI Full 接口以地址映射方式进行数据传输的方法,并且会有效使用片上存储资源进行缓存。

5、掌握系统设计、仿真、综合、验证、上板测试以及调试的方法

三、实验要求

1、实验以个人为单位独立完成。

2、前向传播的计算过程必须采用 Verilog 实现,不允许采用 HLS 方式。

3、实验结束后撰写实验报告,要求包括项目架构设计介绍、关键实验步骤说明、关键部分源代码、设计结果分析及优化思路与技巧。

四、实验环境

1、算法开发框架(TF)

2、Vivado HLx 套装及 Vivado SDK。

3、AX7020 开发板。

4、Mini SD 卡读卡器及 USB 转接线。

五、设计方案

1、总框架设计

a、将助教提供的训练好的模型的参数以及测试样例的数据量化为定点数后以二进制补码的形式存储在 BRAM 中，每次用到时从中读取。

b、将经过仿真、验证后的 LeNet inference 模块以及实例化的几块 BRAM 一起封装成 AXI Lite 接口的 IP 核。AXI Lite 共有 6 个寄存器。其中 slv_reg0 用于向 IP 核传入需要识别的图片序号(0-9)，slv_reg1 用于向 IP 核传输控制信号，控制 LeNet inference 的开始，slv_reg2-slv_reg4 用于存储结果，slv_reg5 用于存储结束信号。

c、流程：

begin

PS 端向 slv_reg0 写入需要识别的图片编号

```
PS 端向 slv_reg1 写入表示开始的控制信号
while PL 进行运算
    PS 端从 slv_reg5 中读出结束信号
    if 结束信号有效
        PS 端读出 slv_reg2-slv_reg4 寄存器中的结果并输出
        break
    end if
end while
end
```

2、定点数量化

a、所有数据和参数被量化为 8 位定点纯小数：1 为符号位 + 7 位小数位。

b、使用的工具是 Octave(Matlab 也可)。

c、量化算法：

```
input: 待量化的数 din, 量化后的小数位数 prec_f
output: 量化后的 8 位二进制补码形式 dout
algrithm:
    beign
        if din == 1 then
            din ::= din * 2^prec_f - 1
        else
            din ::= din * 2^prec_f
        end if
        din ::= round(din)    // 取 din 最近的整数
        if din < 0
            din ::= din + 2^(prec_f+1)
        end if
        dout ::= dec2bin(din, 8)  // 将正整数转化为 8 位二进制形式
    end
```

3、数据缓存区设计

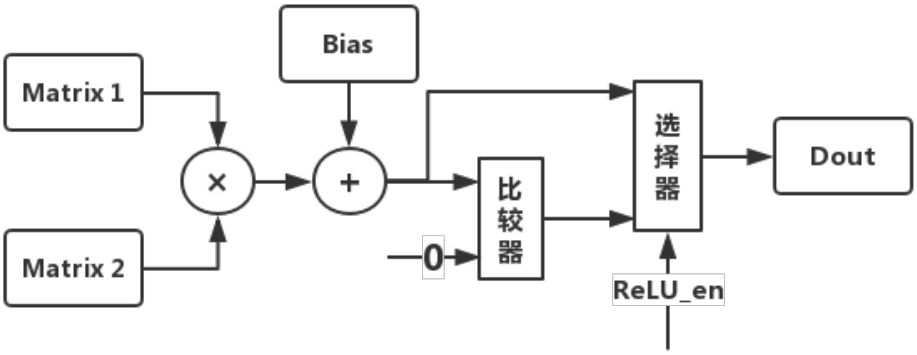
a、使用 PL 端的片上 Block RAM 进行数据存储。开辟三块数据存储区，参数存储区、中间及最后结果存储区和测试样例存储区。

b、参数存储区用于存储训练好的模型的参数，包括 weights 和 bias；中间及最后结果存储区用于存储 LeNet inference 过程中每一层的计算结果已经最后的结果；测试样例存储区用于存储助教提供的 10 张测试图片的数据。

c、以上三个存储区存储时都是按行优先的模式存储的。

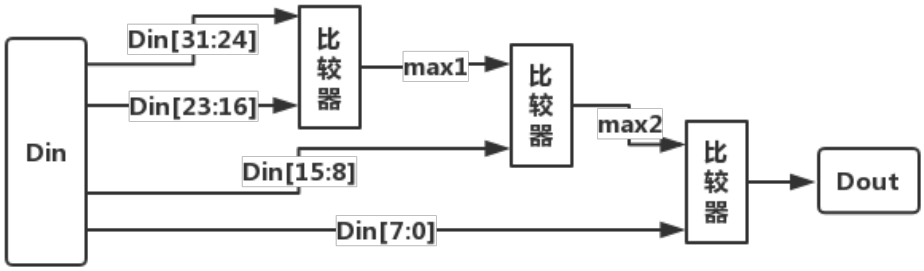
4、数据通路设计

a、25 维向量乘加器



说明：该乘加器模块的输入为 Matrix1、Matrix2，Bias 和 ReLU_en；输出为 Dout。其中 Matrix1 和 Matrix2 为两个 25 维向量，Bias 为参数 bias 或上次计算的结果(具体由状态机控制)，ReLU_en 为是否使用 ReLU 单元的控制信号。该乘加器在卷积层和全连接层中都有使用，在卷积层中用其进行单次卷积运算，在全连接层中用其进行向量的乘加。

b、max pooling

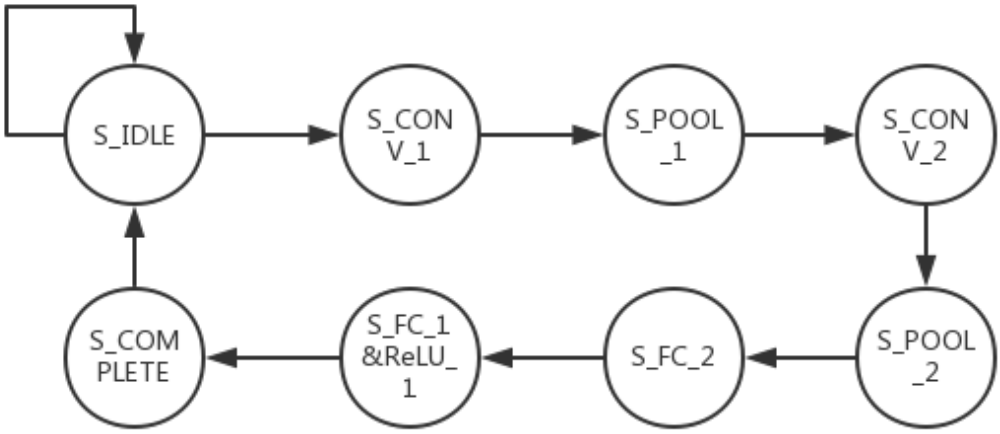


说明：该 Max Pooling 模块的输入为 Din，输出为 Dout。其中，Din 为一个 4 维向量。该模块在池化层使用。

5、控制单元设计

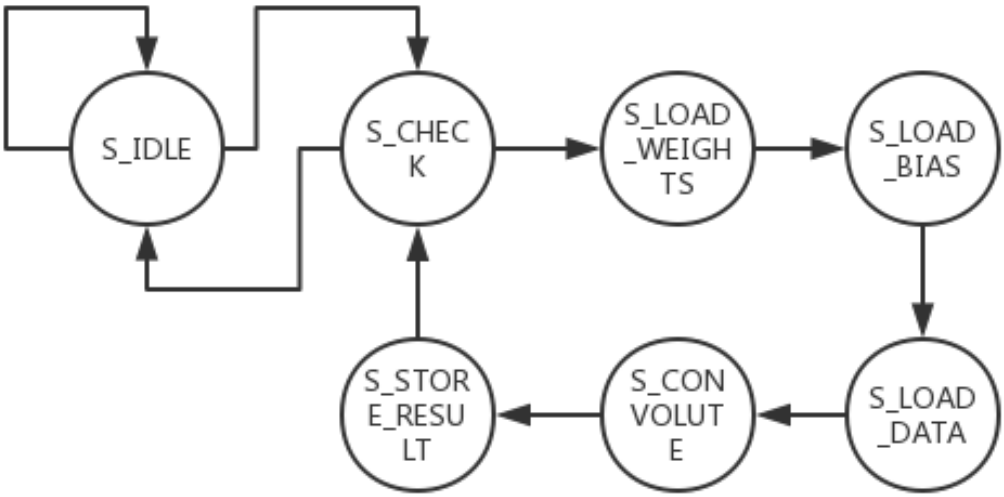
a、顶层状态机

顶层状态机按照实验指导书中所给方案设计：在状态机的每一个状态，完成一层网络的计算。



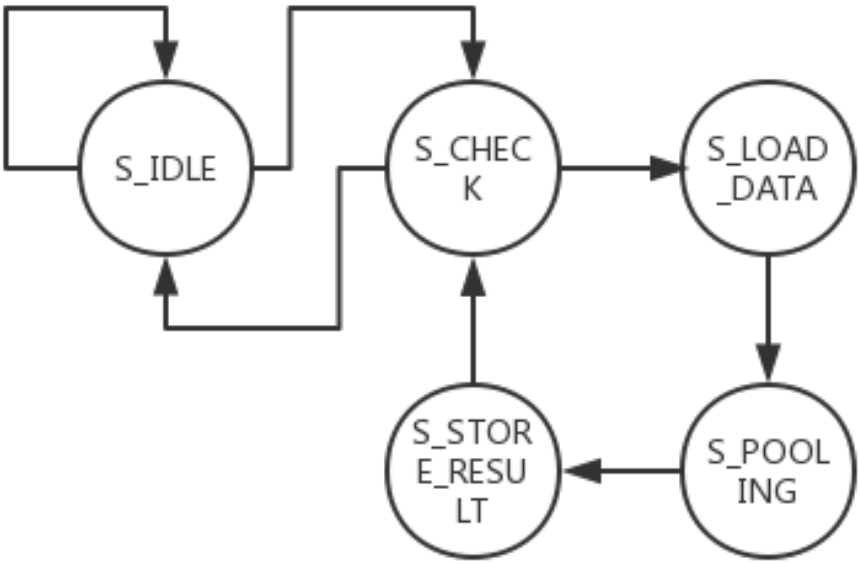
b、卷积层子状态机

以第二个卷积层 conv2 为例，conv1 与其类似：其中 S_CHECK 状态中检查是否已经完成当前层的计算，若已经完成则产生结束信号并转移到 S_IDLE 状态，否则就继续导入参数和数据进行卷积并保存中间结果，直到完成整层网络的计算。



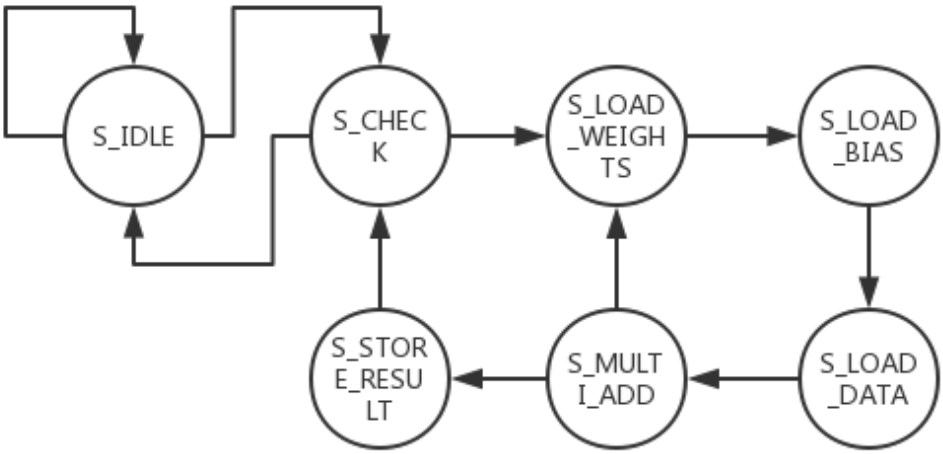
c、池化层子状态机

以第一个池化层 pool1 为例，pool2 与其类似，比卷积层子状态机少了 S_LOAD_WEIGHTS 和 S_LOAD_BIAS 两个状态：



d、全连接层子状态机

以第一个卷积层 fc1 为例，fc2 与其类似，需要注意的就是由于在全连接层复用了 25 维乘加器的数据通路，因此 fc1 中每个 800 维向量一共需要进行 32 次乘加运算，因此每次乘加运算完成后都会根据是否已经算完一个 800 维的向量来决定向哪个状态转移：



六、实现细节

1、定义一系列 LeNet 网络相关的参数

```
// LeNet parameters
`define INPUT_NODE 784
`define OUTPUT_NODE 10
`define IMAGE_SIZE 28
`define NUM_CHANNELS 1
`define NUM_LABELS 10
// Conv1
```

```
`define CONV1_DEEP 20
`define CONV1_SIZE 5
`define CONV1_OUTPUT 24
// Pool1
`define POOL1_INPUT 24
`define POOL1_OUTPUT 12
`define POOL1_SIZE 2
// Conv2
`define CONV2_DEEP 50
`define CONV2_SIZE 5
`define CONV2_OUTPUT 8
// Pool2
`define POOL2_INPUT 8
`define POOL2_OUTPUT 4
`define POOL2_SIZE 2
// Fc1
`define FC1_SIZE 500
// Fc2
`define FC2_SIZE 10
// fixed point
`define DATA_SIZE 8
```

2、定义各块 Block RAM 中各参数存放的基地址，input_bram 里的图片的原始数据就按照图片顺序按行优先存储。

```
// bias_weights_bram
// deep: 4301080
parameter conv1_weights_base = 0,
          conv1_bias_base = 430500,
          conv2_weights_base = 500,
          conv2_bias_base = 430520,
          fc1_weights_base = 25500,
          fc1_bias_base = 430570,
          fc2_weights_base = 425500,
          fc2_bias_base = 431070;

// result_bram
// deep:18910
parameter conv1_result_base = 0,
          pool1_result_base = 11520,
          conv2_result_base = 14400,
          pool2_result_base = 17600,
          fc1_result_base = 18400,
          fc2_result_base = 18900;
```

3、为每个状态机定义其所有的状态，全部采用独热编码的方式，以顶层状态机为例：

```
parameter S_IDLE         = 7'b1000000,
          S_CHECK        = 7'b0100000,
          S_LOAD_WEIGHTS = 7'b0010000,
          S_LOAD_BIAS    = 7'b0001000,
          S_LOAD_DATA    = 7'b0000100,
          S_MULTI_ADD    = 7'b0000010,
          S_STORE_RESULT = 7'b0000001;
```

4、定义各个控制信号，每个状态机都对应一个 en 信号和一个 finish 信号。

5、实现数据通路以及各状态机，最后在顶层模块中将他们连接起来，下面是顶层状态机的核心代码：

```
always@(posedge clk)
begin
    if(rst == 1'b1)
    begin
        state <= S_IDLE;
        //finish <= 1'b1;
        conv_1_en <= 1'b0;
        pool_1_en <= 1'b0;
        conv_2_en <= 1'b0;
        pool_2_en <= 1'b0;
        fc_1_en <= 1'b0;
        relu_1_en <= 1'b0;
        fc_2_en <= 1'b0;
        result_1 <= 0;
        complete_en <= 1'b0;
    end
    else
    begin
        case (state)
            S_IDLE:
            begin
                if(start == 1'b1)
                begin
                    finish <= 1'b0;
                    //result_1 <= 0;
                    conv_1_en <= 1'b1;
                    relu_1_en <= 1'b0;
                    state <= S_CONV_1;
                end
                else
                begin
                    state <= S_IDLE;
                end
            end
            S_CONV_1:
            begin
                if(conv_1_finish == 1'b1)
                begin
                    pool_1_en <= 1'b1;
                    conv_1_en <= 1'b0;
                    relu_1_en <= 1'b0;
                    state <= S_POOL_1;
                end
            end
            S_POOL_1:
            begin
                if(pool_1_finish == 1'b1)
                begin
                    conv_2_en <= 1'b1;
                    pool_1_en <= 1'b0;
                    relu_1_en <= 1'b0;
                    state <= S_CONV_2;
                end
            end
            S_CONV_2:
            begin
```

```
            if(conv_2_finish == 1'b1)
            begin
                pool_2_en <= 1'b1;
                conv_2_en <= 1'b0;
                relu_1_en <= 1'b0;
                state <= S_POOL_2;
            end
        end
        S_POOL_2:
        begin
            if(pool_2_finish == 1'b1)
            begin
                fc_1_en <= 1'b1;
                pool_2_en <= 1'b0;
                relu_1_en <= 1'b1;
                state <= S_FC_1;
            end
        end
        S_FC_1:
        begin
            if(fc_1_finish == 1'b1)
            begin
                fc_2_en <= 1'b1;
                fc_1_en <= 1'b0;
                relu_1_en <= 1'b0;
                state <= S_FC_2;
            end
        end
        S_FC_2:
        begin
            if(fc_2_finish == 1'b1)
            begin
                complete_en <= 1'b1;
                fc_2_en <= 1'b0;
                relu_1_en <= 1'b0;
                state <= COMPLETE;
            end
        end
        COMPLETE:
        begin
            if(complete_finish == 1'b1)
            begin
                result_1 <= tempresult;
                state <= S_IDLE;
                complete_en <= 1'b0;
                finish <= 1'b1;
            end
        end
        default:
        begin
            state <= S_IDLE;
            conv_1_en <= 1'b0;
            pool_1_en <= 1'b0;
            conv_2_en <= 1'b0;
            pool_2_en <= 1'b0;
            fc_1_en <= 1'b0;
            relu_1_en <= 1'b0;
            fc_2_en <= 1'b0;
            complete_en <= 1'b0;
        end
```

```
            endcase
        end
    end
```

6、对最后实现的 lenet_top 模块进行仿真并验证（结果将在后面说明）。将其封装成 AXI Lite 型接口的 IP 核，IP 核的核心代码如下：

```
    always@(posedge S_AXI_ACLK)
    begin
        if(slv_reg1[0] == 1'b1)
        begin
            if(i == 0)
            begin
                rst <= 1'b1;
                i <= i + 1;
                slv_reg5[0] <= 1'b0;
            end
            else if(i == 1)
            begin
                rst <= 1'b0;
                start <= 1'b1;
                graph <= slv_reg0[4:0];
                i <= i + 1;
            end
            else
            begin
                if(i < 4)
                begin
                    i <= i + 1;
                end
                else if(i == 4)
                begin
                    start <= 1'b0;
                    i <= i + 1;
                end
                else
                begin
                    if(lenet_finish == 1'b1)
                    begin
                        slv_reg2 <= result[79:48];
                        slv_reg3 <= result[47:16];
                        slv_reg4[31:16] <= result[15:0];
                        slv_reg5[0] <= 1'b1;
                    end
                end
            end
        end
    end
```

7、进行 Synthesis、Implementation、Create Bitstream 并导出硬件到 SDK 环境中。

这里有几个坑：首先，Synthesis 的时间非常的长；另外，我在 Implementation 时遇到了一个 *critical warning：[Timing 38-282] The design failed to meet the timing requirements.* 在咨询了助教以后得知这个警告是因为时序不满足，可能是因为我的代码中有些**逻辑路径太长**而

导致的。简单的解决办法就是**降频**；复杂一点的办法可以是将其**流水化**。在这里我采用原理降频的做法。**具体做法如下：**a、若工程自动添加了.xdc 约束文件，则可以在约束文件中改；b、否则，打开 Block Design，双击配置 Zynq，在 Clock Configuration 中的 PL Fabric Clocks 下修改 FCLK_CLK0 的 Required Frequency，我将原来的 50 改成了 25，顺利完成了 Implementation。

8、SDK 中的核心 C 代码如下：

```
    LENET_mWriteReg(XPAR_LENET_0_S00_AXI_BASEADDR, 0, 0x00000009);
    LENET_mWriteReg(XPAR_LENET_0_S00_AXI_BASEADDR, 4, 0x00000001);

     u32 r0,r1,r2,r3;
     while(1){
          r0 = LENET_mReadReg(XPAR_LENET_0_S00_AXI_BASEADDR, 20);
          if(r0 == 0x00000001){
                    r1 = LENET_mReadReg(XPAR_LENET_0_S00_AXI_BASEADDR, 8);
                    r2 = LENET_mReadReg(XPAR_LENET_0_S00_AXI_BASEADDR, 12);
                    r3 = LENET_mReadReg(XPAR_LENET_0_S00_AXI_BASEADDR, 16);
                    break;
          }
     }
     xil_printf("r1 = %08x, r2 = %08x, r3 = %08x  \n\r", r1, r2, r3);
```

9、关于仿真验证过程中的调试

由于时间关系，本来想用 octave 实现每一层的定点仿真的，最后并没能做完。关于中间结果也只是在仿真过程中将其取出几个，然后和自己用 octave 算的结果进行对照，虽然对照的结果都是正确的，但感觉不是很严谨，因此并没有代码保留下来。

10、数据预处理阶段的代码(octave 定点化代码)，可能看起来有些乱，因为助教提供的训练好的参数的文件中的格式不太一样，因此有些地方需要手动修改代码，源代码 fixed.m 已和工程一同提交。

```
load temp.mat
input = temp;
%for i = 1 : length(input)
%    input(i) = input(i) * 1.00000e-02;
%end
%input
s = size(temp)
prec_f = 7;
for i = 1 : s(1)
     for j = 1 : s(2)
          if (input(i, j) == 1)
               data_out(i, j) = input(i, j) * pow2(prec_f) - 1;
          else
              data_out(i, j) = input(i, j) * pow2(prec_f);
          end
     end
end
data_out = round(data_out);
for i = 1 : s(1)
```

```
    for j = 1 : s(2)
        if(data_out(i, j) < 0)
            data_out(i, j) = data_out(i, j) + pow2(prec_f + 1);
        end
        col = (j-1)*9 + 1;
        data_out_bin(i,col:col+7) = dec2bin(data_out(i, j),8);
        data_out_bin(i,col+8) = ' ';
     end
end
size(data_out_bin)
save result data_out_bin
```

七、实验结果

最终的实验结果不是那么令人满意，助教提供的 10 张测试图片中，只有 第 7 张图是识别成功的，其他都失败了。由于输出的是 16 进制的，因此我还将每个结果进行了处理，转化为二进制，具体的结果文件为 sdkresult.mat，已和工程一同提交。

八、分析与讨论

我认为对我实验结果产生影响比较大的主要有两个方面：**一个是定点数的设计**，设置成定点纯小数可能不太合适，导致中间结果整数位溢出；**另一个是全连接层中 25 维向量乘加器的复用**，因为每次定点数乘加结束以后都有一个移位操作，因此也会造成精度的损失，我觉得可以尝试更加平衡的乘加器设计，不过也只有试了才知道。

九、参考资料

matlab 量化浮点数:
http://blog.csdn.net/shichaog/article/details/50809478
FIR 滤波器的 verilog 实现（主要是学习了一下他的实现步骤）:
http://www.bijishequ.com/detail/441489?p=57-57
Xilinx block ram 研究验证详解:
http://www.openhw.org/module/forum/thread-657945-1-1.html
浮点数定点数数据格式:
https://www.jianshu.com/p/ef2211ca0e88
tensorflow 官网:
https://www.tensorflow.org/