实验三：在 Zynq FPGA 上实现 VGG16 网络
实验报告

姓名：张安澜
学号：15061075
2018 年 1 月 27 日

小组成员与分工

成员：张安澜、游心、胡卓仑

分工：所有工作均由我（张安澜）一人完成，主要原因是队友不配合，也没有想做的决心。在这里感谢司靖辉同学以及助教学长在我完成实验的过程中给我提供的帮助。

一、摘要

在这个实验中，我在 Znqy 7020 板子上实现了一个大型 CNN 网络 VGG16 的前向传播过程。并进行了对于优化的思考。

二、实验目的

设计专门硬件结构,实现 VGG16 网络的前向传播过程,进行图片识别(inference):

1. 使用算法开发框架(Cafffe/TF/PyTorch/etc.)进行训练、测试,理解 VGG16 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 xilffs 库进行 SD 卡读写方法(参考实验附件二中 SD 卡读写部分)、使用 DMA 进行 PS、PL 端数据交互的方法等。
3. 掌握进行分层前向传播计算的结构的设计方法。
4. 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. 实验以小组为单位共同完成,每组 2~3 人。
2. 实验内容: PS 端将训练好的模型及测试数据读入,PL 端实现前向传播的计算过程,模型权值存入 SD 卡,通过 DMA 传输至 PL 端的片上存储(BRAM),PS-PL 通过 AXI 接口连接,生成硬件并导入 SDK 环境中,经过仿真、综合、验证之后,下载到开发板上进行实测。
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。并将完整的工程文件提交到课程中心,助教会对工程设计的源码等内容进行查重,并进行上板实测。

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 SD 卡中，通过 Xilinx 提供的简易 fat32 文件系统库 Xilffs 进行读取，并采用 DMA 实现 PS 与 PL 端数据交互。

b、设计时的思考

（1）每层网络的计算都分为 loaddma、vgg16_calculate 和 storedma 三大状态；

（2）loaddma 表示从 PS 端向 PL 端传输数据，vgg16_calculate 表示 PL 端进行相应的运算，storedma 表示从 PL 端向 PS 端传输数据；

（3）设计时采用了简化的思想，主要体现在：实例化了一块超大的 Block RAM，先将每次计算用到的所有数据都从外面的小的 Block RAM 中存到大的这块 Block RAM 中之后才进行相应的运算，因此外面的小 Block RAM 需要双端口模式；不管 PL 端寄存器资源是否够用，在必要时都将结果存回内存， 这样主要是为了实现时写代码方便；

（4）由于网络太大，必须对每一层进行拆分，而拆分时势必涉及到根据通道数进行拆分，比如原来是 64 通道的原始数据要拆成 16 个 4 通道的原始数据，因此在 PS 和 PL 端必须要有一个信号 addbefore 来控制是否需要和之前计算的结果相加；而每次导入 bias 也会根据计算的过程进行改变，这个在 PS 端中写代码控制即可；

（5）由于 VGG16 这个 IP 核中的 Block RAM 需要从外面的小的 Block RAM 中读取数据或向外面的小的 Block RAM 写入数据，因此需要 PS 端给它一个控制信号 base_addr 来控制读到的数据要从这块大的 Block RAM 中的哪一个地址开始保存，或是将这块大的 Block RAM 中的哪一个地址开始的数据写入外面小的 Block RAM 中，以及另一个信号 num 表示读入或写入的数据个数，当然还有 PS 和 PL 端交互的控制信号 loaddma_start、

l oaddma_finish、storedma_start、storedma_finish；

（6）同时，PS 端还要控制 PL 端的计算，之前已经提到了一个 addbefore 信号，另外 PS 端还需要传入一个 layer 信号表示当前需要计算的是第几层，具体的编号方式在 d 中给出，总之 PL 端读到这个这个 layer 信号就会进行对应的运算，当然还有 PS 和 PL 端交互的控制信号 vgg16_start、vgg16_finish；

（7）基于（5）、（6），在封装 IP 时，需要 6 个寄存器，具体在 c 中说明。同时还需要添加一些 User Ports：doutb_in、clkb_out、rstb_out、enb_out、web_out、addrb_out、dinb_out 来和外面的小 Block RAM 的其中一个端口进行连接；

（8）对网络的每一层，PS 端每次都会直接先把该层用到的所有参数全部从 SD 卡中读入内存（经过尝试应该是可以存下的），然后根据计算的具体情况将对应的数据和参数从 PS 端传输至 PL 端，由于 PS 和 PL 之间传输的数据类型是 u32，而从 SD 卡中读取时的数据类型是 u8，因此中间还需要进行一系列的转换，并且根据具体计算情况选择对应的数据和参数也不是一件简单的事，因此这部分实际上是该实验中比较麻烦的一部分；

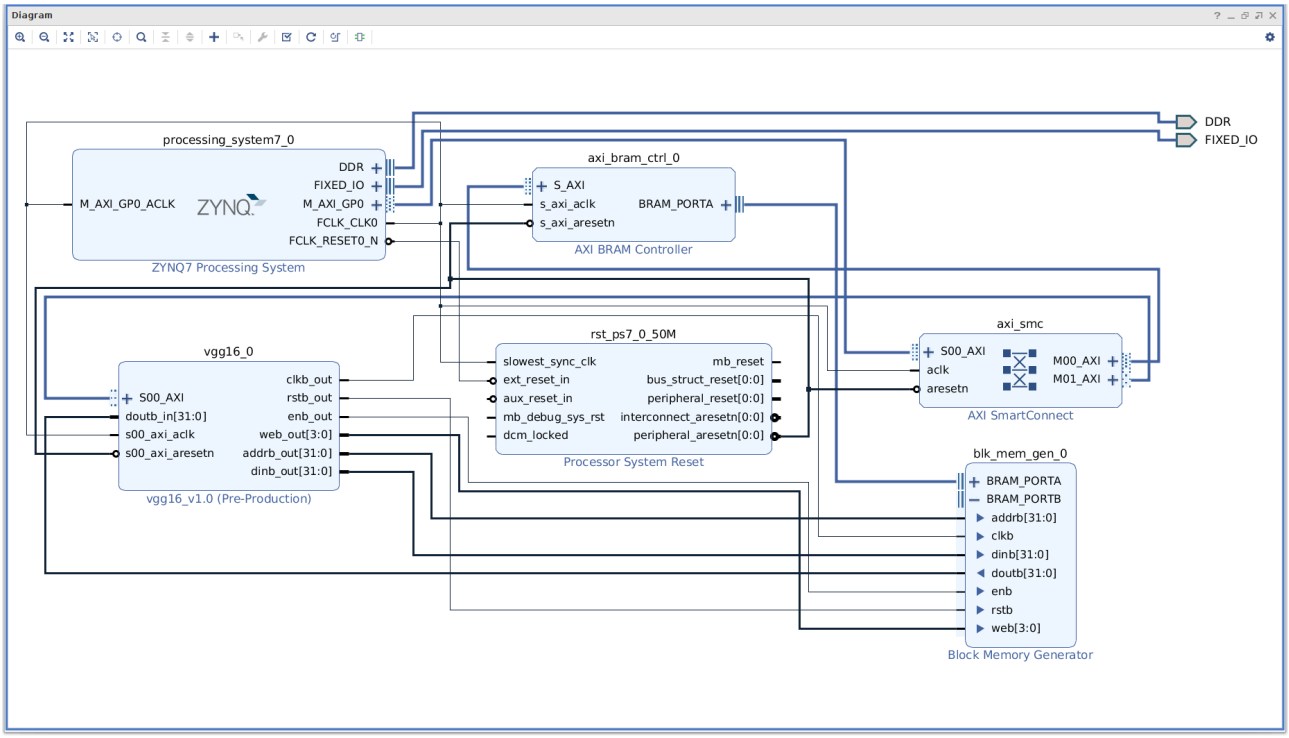
（9）为了简化处理，所有从 PL 端传回 PS 端的中间结果，都不写入 SD 卡中，只有最后结果写入 SD 卡；

（10）每次 PS 端和 PL 端之间的数据传输也需要做好设计，因为不是一次性能传完的，以及每次 PL 端运算时数据、参数和结果保存的基地址，这些都需要提前算好以参数的形式写在代码中；

（11）为了追求实现的简单，暂时先不考虑流水、模型压缩、参数共享等优化了，至少先能在板子上实现这个网络再说；

c、将经过仿真、验证后的 VGG16 inference 模块以及实例化的 1 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 核传输控制信号 vgg16_start(slv_reg0[0])、loaddma_start(slv_reg0[1])、storedma_start(slv_reg0[2])，slv_reg1 用

于存储 IP 核传出的结束信号 vgg16_finish(slv_reg1[0])、 loaddma_finish(slv_reg1[1])、storedma_finish(slv_reg1[2])，slv_reg2 用于传输 addbefore 信号(sl_reg2[0])，slv_reg3 用于传输 layer 信号(slv_reg3[4:0])，slv_reg4 用于传输 base_addr 信号(slv_reg4[19:0])，slv_reg5 用于传输 num 信号(slv_reg5[31:0])，上述信号的意义在 b 中已经提到，最后，总的 block design 如下：



d、顶层逻辑：

从 Conv1_1 开始按顺序对网络的层次进行编号，共编为 0 到 20 号（包括池化层）

for layercount: 0 to 21

begin

PS 端从 SD 卡中读入该层的参数（以及第一层需要读原始数据）

if 卷积层

for 每一层拆分出的小卷积

begin

PS 端从内存中选择当前需要的数据和参数通过 DMA 传输到 PL 端对当前的小型网络进行运算

if 对某些 filter 已经做完所有通道的运算

将结果通过 DMA 从 PL 端传输到 PS 端，保存在内存中

end if

end for

end if

if 池化层

for 每一层拆分出的小池化

begin

PS 端从内存中选择当前需要的数据通过 DMA 传输到 PL 端对当前的小型网络进行运算

```
                将结果通过 DMA 从 PL 端传输到 PS 端，保存在内存中
            end for
        end if
        if 全连接层
            for 每一层拆分出的小全连接
                PS 端从内存中选择当前需要的数据通过 DMA 传输到 PL 端
                对当前的小型网络进行运算
                将结果通过 DMA 从 PL 端传输到 PS 端，保存在内存中
            end for
            if 全连接层 3 已经完成
                将内存中全连接层 3 的结果写入 SD 卡的文件中
                exit(0)
            end if
        end if
    end for
```

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 位符号位 + 1 位整数位 + 6 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
        douta ::= dec2bin(din, 7) // 将正整数转化为 7 位二进制形式
        dout ::= {douta[0], douta}
    end
```

3、VGG16 网络结构的分析与拆分

a、VGG16 的计算花费

Layer name	data	weight	bias	result
conv1_1	224*224*3	3*3*3*64	64	224*224*64
conv1_2	224*224*64	3*3*64*64	64	224*224*64
pool1	224*224*64	0	0	112*112*64

conv2_1	112*112*64	3*3*64*128	128	112*112*128
conv2_2	112*112*128	3*3*128*128	128	112*112*128
pool2	112*112*128	0	0	56*56*128
conv3_1	56*56*128	3*3*128*256	256	56*56*256
conv3_2	56*56*256	3*3*256*256	256	56*56*256
conv3_3	56*56*256	3*3*256*256	256	56*56*256
pool3	56*56*256	0	0	28*28*256
conv4_1	28*28*256	3*3*256*512	512	28*28*512
conv4_2	28*28*512	3*3*512*512	512	28*28*512
conv4_3	28*28*512	3*3*512*512	512	28*28*512
pool4	28*28*512	0	0	14*14*512
conv5_1	14*14*512	3*3*512*512	512	14*14*512
conv5_2	14*14*512	3*3*512*512	512	14*14*512
conv5_3	14*14*512	3*3*512*512	512	14*14*512
pool5	14*14*512	0	0	7*7*512
fc1	7*7*512	4096*7*7*512	4096	4096
fc2	4096	4096*4096	4096	4096
fc3	4096	1000*4096	1000	1000

b、每层网络的拆分

拆分的原则：在 Block RAM 资源充足的情况下最大化 Block RAM 的使用，且保证每一层网络拆分后的小网络的结构相同，拆分后得到的每一层拆分的小网络的参数和数据，以及需要计算的次数如下：

Layer name	data	weight	bias	result	count
conv1_1	224*224*3	3*3*3*4	4	224*224*4	1*16
conv1_2	224*224*4	3*3*4*4	4	224*224*4	16*16
pool1	224*224*4	0	0	112*112*4	16
conv2_1	112*112*16	3*3*16*16	16	112*112*16	4*8
conv2_2	112*112*16	3*3*16*16	16	112*112*16	8*8
pool2	112*112*16	0	0	56*56*16	8
conv3_1	56*56*64	3*3*64*64	64	56*56*64	2*4
conv3_2	56*56*64	3*3*64*64	64	56*56*64	4*4
conv3_3	56*56*64	3*3*64*64	64	56*56*64	4*4
pool3	56*56*64	0	0	28*28*64	4

conv4_1	28*28*256	3*3*256*64	64	28*28*64	1*8
conv4_2	28*28*256	3*3*256*64	64	28*28*64	2*8
conv4_3	28*28*256	3*3*256*64	64	28*28*64	2*8
pool4	28*28*256	0	0	14*14*256	2
conv5_1	14*14*512	3*3*512*64	64	14*14*64	1*8
conv5_2	14*14*512	3*3*512*64	64	14*14*64	1*8
conv5_3	14*14*512	3*3*512*64	64	14*14*64	1*8
pool5	14*14*512	0	0	7*7*512	1
fc1	7*7*512	16*7*7*512	16	16	256
fc2	4096	128*4096	128	128	32
fc3	4096	125*4096	125	125	8

4、数据缓存区设计

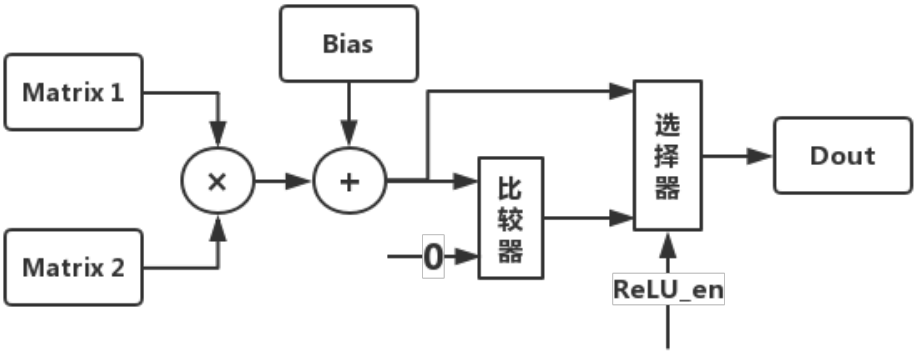
a、为了通过 DMA 在 PS 与 PL 端传输数据，以 BRAM Controller 模式开辟一块 **32*2028** 大小的双端口 Block RAM 用于共享数据，称为 bram1；

b、为了简化处理，在 VGG16 这个 IP 核内部还实例化了一块大小为 **8*528640（该大小由 2 中第二个表格的 fc2 层拆分的小网络计算得到）**的单端口 Block RAM 用于保存每次计算用到的数据和参数以及计算的结果，每次计算时数据、参数和结果的基地址都是提前计算好作为 parameter 写在代码中，数据和参数从 bram1 中导入，称为 bram2，只实例化一块 Block RAM 的原因是有的网络计算时数据量远大于参数量，也有的参数量远大于数据量，因此若按原始数据、参数和结果分别实例化一块 Block RAM 会导致利用率降低；

c、以上 Block RAM 存储时都是按行优先的模式存储的，综合后报告中显示用掉了 **94%** 的 Block RAM 资源。

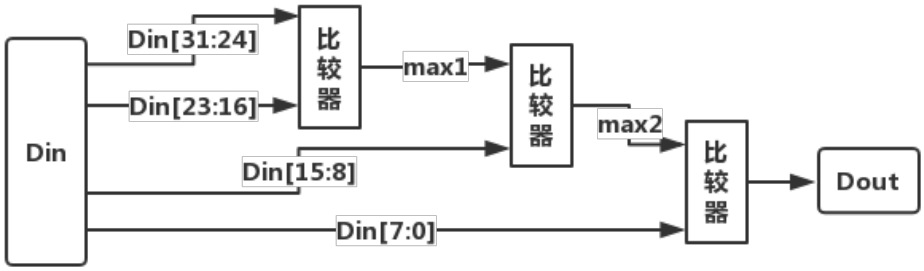
5、数据通路设计

a、9 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 9 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

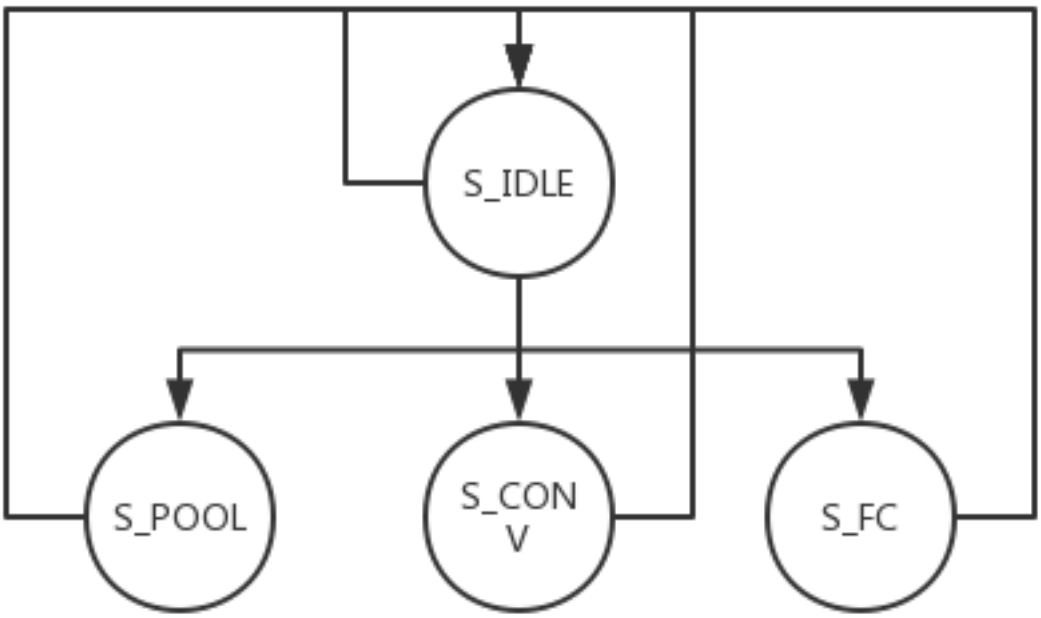


说明：该 Max Pooling 模块的输入为 Din，输出为 Dout。其中，Din 为一个 4 维向量。该模块在池化层使用。

6、控制单元设计

a、VGG16 顶层模块状态机

VGG16 顶层模块状态机设计如下：可以理解成当 PS 端有计算的信号传输过来时，顶层模块根据传输过来的 layer 参数跳转到对应的状态进行运算，运算结束以后回到 S_IDLE 状态同时向 PS 端传输结束信号，并等待下一次运算信号的传输。这和我之前设计的 LeNet 的顶层模块状态机有很大的区别，主要是因为这次由于数据和参数要从 PS 端传输至 PL 端，因此总的控制逻辑在 PS 端，VGG16 的 IP 核只是将卷积、池化和全连接进行了封装，因此只要向其传入对应信号即可进行对应的运算。这也是我觉得我比实验二中设计上进步的一点。



b、卷积层子状态机

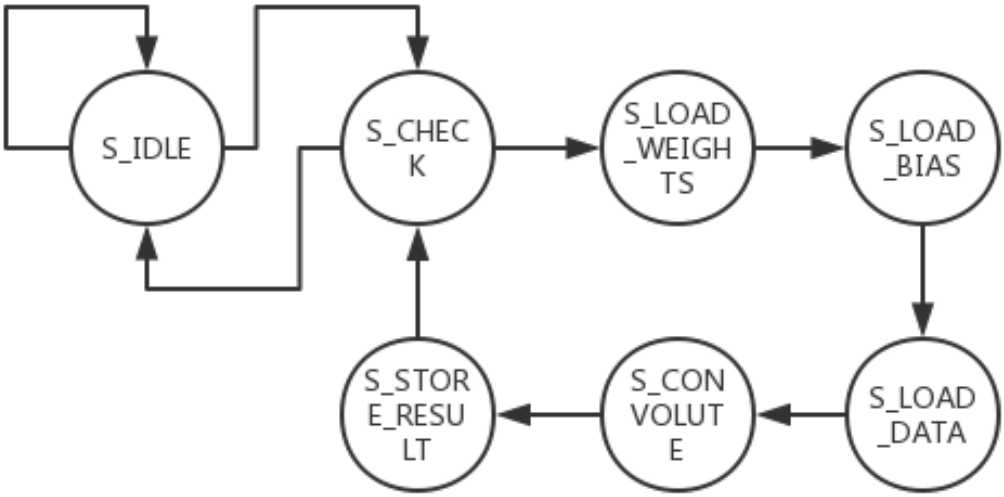
卷积层子状态机设计如下：其中 S_CHECK 状态中检查是否已经完成当前层的计算，若已经完成则产生结束信号并转移到 S_IDLE 状态，否则就继续导入参数和数据进行卷积并保存中间结果，直到完成整层网络的计算。

根据实验要求和 VGG16 网络的特点对实验二中的卷积层子状态机进行了修改，得到了该卷积层子状态机，有如下几个方面的修改：

（1）实现了根据寄存器配置完成不同行数、列数、通道数、filter 数的卷积运算；

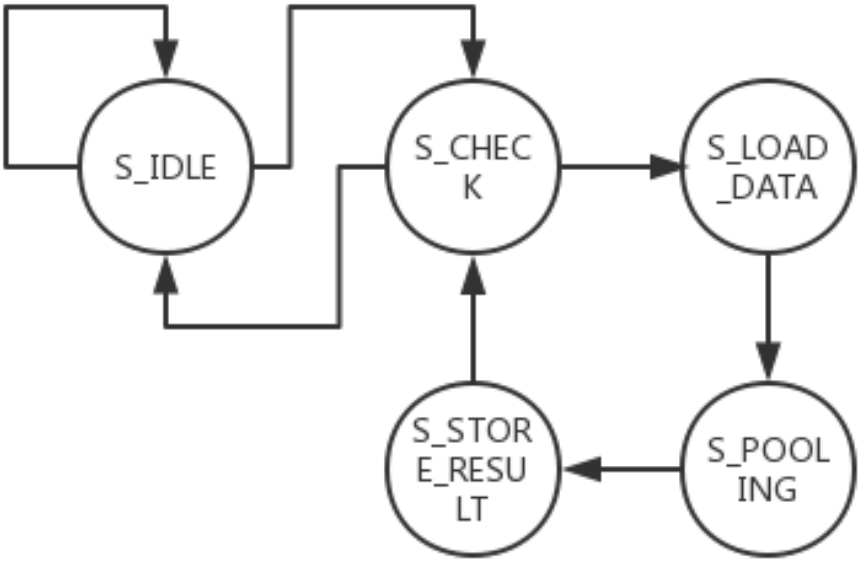
（2）将原来的没有 padding 的卷积运算改为了有 padding 的卷积运算；

（3）由于对某些输入数据的通道数特别多的层根据通道数进行了拆分，因此在 S_STORE_RESULT 状态中加入相应判断，看是否需要和之前计算出来的值相加（由于每一层拆分后的每个小网络的结构都是一样的，因此计算出的结果的保存的地址范围也是一样的）



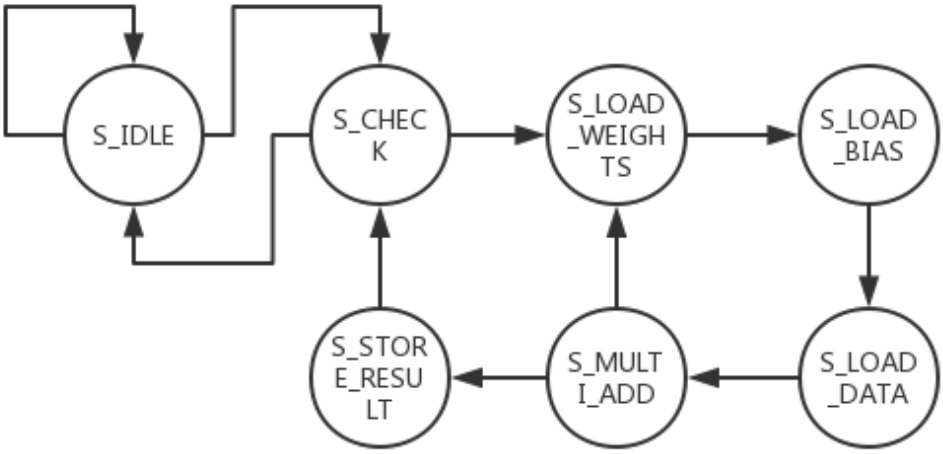
c、池化层子状态机

池化层子状态机设计如下：同样也实现了根据寄存器配置完成不同行数、列数的池化运算；



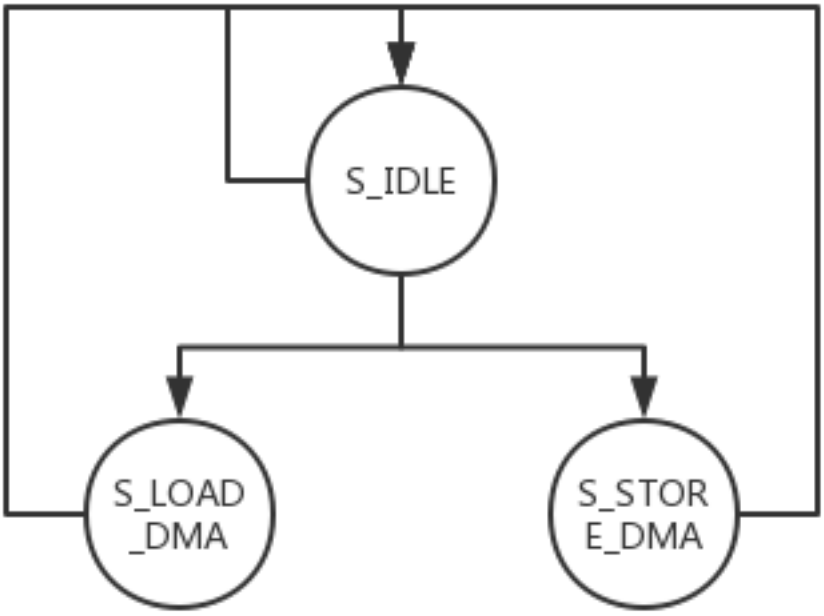
d、全连接层子状态机

全连接层子状态机设计如下，需要注意的就是由于在全连接层复用了 9 维乘加器的数据通路，因此每次乘加运算完成后都会根据是否已经算完一个向量来决定向哪个状态转移：同样实现了根据寄存器配置完成不同行数、列数、结果特征数的全连接运算；



e、IP 核内部 Block RAM 状态机

设计如下：理解方式和 VGG16 顶层模块的状态机差不多，都是在 S_IDLE 状态等待 PS 端传入的控制信号（loaddma_start 或 storedma_start）来决定往哪个状态转移的，结束后又回到 S_IDLE 状态等待之后的信号。



六、实现细节

1、PL 端定义每次计算用到的数据、参数和结果保存的基地址

```
    parameter conv1_1_data_base_addr   = 0,
              conv1_1_weight_base_addr = 150528,
              conv1_1_bias_base_addr   = 150636,
              conv1_1_result_base_addr = 150640;
```

```
    parameter conv1_2_data_base_addr   = 0,
              conv1_2_weight_base_addr = 200704,
              conv1_2_bias_base_addr   = 200848,
              conv1_2_result_base_addr = 200852;

    parameter conv2_data_base_addr   = 0,
              conv2_weight_base_addr = 200704,
              conv2_bias_base_addr   = 203008,
              conv2_result_base_addr = 203024;

    parameter conv3_data_base_addr   = 0,
              conv3_weight_base_addr = 200704,
              conv3_bias_base_addr   = 237568,
              conv3_result_base_addr = 237632;

    parameter conv4_data_base_addr   = 0,
              conv4_weight_base_addr = 200704,
              conv4_bias_base_addr   = 348160,
              conv4_result_base_addr = 348224;

    parameter conv5_data_base_addr   = 0,
              conv5_weight_base_addr = 100352,
              conv5_bias_base_addr   = 395264,
              conv5_result_base_addr = 395328;

    parameter pool1_data_base_addr   = 0,
              pool1_result_base_addr = 200704;

    parameter pool2_data_base_addr   = 0,
              pool2_result_base_addr = 200704;

    parameter pool3_data_base_addr   = 0,
              pool3_result_base_addr = 200704;

    parameter pool4_data_base_addr   = 0,
              pool4_result_base_addr = 200704;

    parameter pool5_data_base_addr   = 0,
              pool5_result_base_addr = 100352;

    parameter fc1_data_base_addr   = 0,
              fc1_weight_base_addr = 25088,
              fc1_bias_base_addr   = 426496,
              fc1_result_base_addr = 426512;

    parameter fc2_data_base_addr   = 0,
              fc2_weight_base_addr = 4096,
              fc2_bias_base_addr   = 528384,
              fc2_result_base_addr = 528512;

    parameter fc3_data_base_addr   = 0,
              fc3_weight_base_addr = 4096,
              fc3_bias_base_addr   = 516096,
              fc3_result_base_addr = 516221;
```

2、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    = 22'b1000_0000_0000_0000_0000_00,
              S_CONV1_1 = 22'b0100_0000_0000_0000_0000_00,
```

```
                S_CONV1_2  = 22'b0010_0000_0000_0000_0000_00,
                S_POOL1    = 22'b0001_0000_0000_0000_0000_00,
                S_CONV2_1  = 22'b0000_1000_0000_0000_0000_00,
                S_CONV2_2  = 22'b0000_0100_0000_0000_0000_00,
                S_POOL2    = 22'b0000_0010_0000_0000_0000_00,
                S_CONV3_1  = 22'b0000_0001_0000_0000_0000_00,
                S_CONV3_2  = 22'b0000_0000_1000_0000_0000_00,
                S_CONV3_3  = 22'b0000_0000_0100_0000_0000_00,
                S_POOL3    = 22'b0000_0000_0010_0000_0000_00,
                S_CONV4_1  = 22'b0000_0000_0001_0000_0000_00,
                S_CONV4_2  = 22'b0000_0000_0000_1000_0000_00,
                S_CONV4_3  = 22'b0000_0000_0000_0100_0000_00,
                S_POOL4    = 22'b0000_0000_0000_0010_0000_00,
                S_CONV5_1  = 22'b0000_0000_0000_0001_0000_00,
                S_CONV5_2  = 22'b0000_0000_0000_0000_1000_00,
                S_CONV5_3  = 22'b0000_0000_0000_0000_0100_00,
                S_POOL5    = 22'b0000_0000_0000_0000_0010_00,
                S_FC1      = 22'b0000_0000_0000_0000_0001_00,
                S_FC2      = 22'b0000_0000_0000_0000_0000_10,
                S_FC3      = 22'b0000_0000_0000_0000_0000_01;
```

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：其实后来写完状态机之后发现很多状态都可以合并成一个状态，但是由于时间关系没有进行修改。

```
always@(posedge clk)
 begin
        if(rst == 1'b1)
        begin
            state <= S_IDLE;
            conv_en <= 1'b0;
            pool_en <= 1'b0;
            fc_en <= 1'b0;
        end
        else
        begin
            case (state)
                S_IDLE:
                begin
                    if(vgg16_start == 1'b1)
                    begin
                        vgg16_finish <= 1'b0;
                        case (layer)
                            5'b00000: // 0  conv1_1
                            begin
                                conv_en <= 1'b1;
                                weight_base_addr <= conv1_1_weight_base_addr;
                                bias_base_addr <= conv1_1_bias_base_addr;
                                data_base_addr <= conv1_1_data_base_addr;
                                result_base_addr <= conv1_1_result_base_addr;
                                rownum <= 10'd224;
                                columnnum <= 10'd224;
                                channelnum <= 10'd3;
                                filternum <= 10'd4;
                                state <= S_CONV1_1;
```

```
                end
                5'b00001: // 1  conv1_2
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv1_2_weight_base_addr;
                    bias_base_addr <= conv1_2_bias_base_addr;
                    data_base_addr <= conv1_2_data_base_addr;
                    result_base_addr <= conv1_2_result_base_addr;
                    rownum <= 10'd224;
                    columnnum <= 10'd224;
                    channelnum <= 10'd4;
                    filternum <= 10'd4;
                    state <= S_CONV1_2;
                end
                5'b00010: // 2  pool1
                begin
                    pool_en <= 1'b1;
                    rownum <= 10'd112;
                    columnnum <= 10'd112;
                    channelnum <= 10'd4;
                    data_base_addr <= pool1_data_base_addr;
                    result_base_addr <= pool1_result_base_addr;
                    state <= S_POOL1;
                end
                5'b00011: // 3  conv2_1
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv2_weight_base_addr;
                    bias_base_addr <= conv2_bias_base_addr;
                    data_base_addr <= conv2_data_base_addr;
                    result_base_addr <= conv2_result_base_addr;
                    rownum <= 10'd112;
                    columnnum <= 10'd112;
                    channelnum <= 10'd16;
                    filternum <= 10'd16;
                    state <= S_CONV2_1;
                end
                5'b00100: // 4  conv2_2
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv2_weight_base_addr;
                    bias_base_addr <= conv2_bias_base_addr;
                    data_base_addr <= conv2_data_base_addr;
                    result_base_addr <= conv2_result_base_addr;
                    rownum <= 10'd112;
                    columnnum <= 10'd112;
                    channelnum <= 10'd16;
                    filternum <= 10'd16;
                    state <= S_CONV2_2;
                end
                5'b00101: // 5  pool2
                begin
                    pool_en <= 1'b1;
                    rownum <= 10'd56;
                    columnnum <= 10'd56;
                    channelnum <= 10'd16;
                    data_base_addr <= pool2_data_base_addr;
                    result_base_addr <= pool2_result_base_addr;
                    state <= S_POOL2;
                end
```

```
                5'b00110: // 6  conv3_1
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv3_weight_base_addr;
                    bias_base_addr <= conv3_bias_base_addr;
                    data_base_addr <= conv3_data_base_addr;
                    result_base_addr <= conv3_result_base_addr;
                    rownum <= 10'd56;
                    columnnum <= 10'd56;
                    channelnum <= 10'd64;
                    filternum <= 10'd64;
                    state <= S_CONV3_1;
                end
                5'b00111: // 7  conv3_2
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv3_weight_base_addr;
                    bias_base_addr <= conv3_bias_base_addr;
                    data_base_addr <= conv3_data_base_addr;
                    result_base_addr <= conv3_result_base_addr;
                    rownum <= 10'd56;
                    columnnum <= 10'd56;
                    channelnum <= 10'd64;
                    filternum <= 10'd64;
                    state <= S_CONV3_2;
                end
                5'b01000: // 8  conv3_3
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv3_weight_base_addr;
                    bias_base_addr <= conv3_bias_base_addr;
                    data_base_addr <= conv3_data_base_addr;
                    result_base_addr <= conv3_result_base_addr;
                    rownum <= 10'd56;
                    columnnum <= 10'd56;
                    channelnum <= 10'd64;
                    filternum <= 10'd64;
                    state <= S_CONV3_3;
                end
                5'b01001: // 9  pool3
                begin
                    pool_en <= 1'b1;
                    rownum <= 10'd28;
                    columnnum <= 10'd28;
                    channelnum <= 10'd64;
                    data_base_addr <= pool3_data_base_addr;
                    result_base_addr <= pool3_result_base_addr;
                    state <= S_POOL3;
                end
                5'b01010: // 10 conv4_1
                begin
                    conv_en <= 1'b1;
                    weight_base_addr <= conv4_weight_base_addr;
                    bias_base_addr <= conv4_bias_base_addr;
                    data_base_addr <= conv4_data_base_addr;
                    result_base_addr <= conv4_result_base_addr;
                    rownum <= 10'd28;
                    columnnum <= 10'd28;
                    channelnum <= 10'd256;
                    filternum <= 10'd64;
```

```
                state <= S_CONV4_1;
            end
            5'b01011: // 11 conv4_2
            begin
                conv_en <= 1'b1;
                weight_base_addr <= conv4_weight_base_addr;
                bias_base_addr <= conv4_bias_base_addr;
                data_base_addr <= conv4_data_base_addr;
                result_base_addr <= conv4_result_base_addr;
                rownum <= 10'd28;
                columnnum <= 10'd28;
                channelnum <= 10'd256;
                filternum <= 10'd64;
                state <= S_CONV4_2;
            end
            5'b01100: // 12 conv4_3
            begin
                conv_en <= 1'b1;
                weight_base_addr <= conv4_weight_base_addr;
                bias_base_addr <= conv4_bias_base_addr;
                data_base_addr <= conv4_data_base_addr;
                result_base_addr <= conv4_result_base_addr;
                rownum <= 10'd28;
                columnnum <= 10'd28;
                channelnum <= 10'd256;
                filternum <= 10'd64;
                state <= S_CONV4_3;
            end
            5'b01101: // 13 pool4
            begin
                pool_en <= 1'b1;
                rownum <= 10'd14;
                columnnum <= 10'd14;
                channelnum <= 10'd256;
                data_base_addr <= pool4_data_base_addr;
                result_base_addr <= pool4_result_base_addr;
                state <= S_POOL4;
            end
            5'b01110: // 14 conv5_1
            begin
                conv_en <= 1'b1;
                weight_base_addr <= conv5_weight_base_addr;
                bias_base_addr <= conv5_bias_base_addr;
                data_base_addr <= conv5_data_base_addr;
                result_base_addr <= conv5_result_base_addr;
                rownum <= 10'd14;
                columnnum <= 10'd14;
                channelnum <= 10'd512;
                filternum <= 10'd64;
                state <= S_CONV5_1;
            end
            5'b01111: // 15 conv5_2
            begin
                conv_en <= 1'b1;
                weight_base_addr <= conv5_weight_base_addr;
                bias_base_addr <= conv5_bias_base_addr;
                data_base_addr <= conv5_data_base_addr;
                result_base_addr <= conv5_result_base_addr;
                rownum <= 10'd14;
                columnnum <= 10'd14;
```

```
                channelnum <= 10'd512;
                filternum <= 10'd64;
                state <= S_CONV5_2;
            end
            5'b10000:  // 16 conv5_3
            begin
                conv_en <= 1'b1;
                weight_base_addr <= conv5_weight_base_addr;
                bias_base_addr <= conv5_bias_base_addr;
                data_base_addr <= conv5_data_base_addr;
                result_base_addr <= conv5_result_base_addr;
                rownum <= 10'd14;
                columnnum <= 10'd14;
                channelnum <= 10'd512;
                filternum <= 10'd64;
                state <= S_CONV5_3;
            end
            5'b10001:  // 17 pool5
            begin
                pool_en <= 1'b1;
                rownum <= 10'd7;
                columnnum <= 10'd7;
                channelnum <= 10'd512;
                data_base_addr <= pool5_data_base_addr;
                result_base_addr <= pool5_result_base_addr;
                state <= S_POOL5;
            end
            5'b10010:  // 18 fc1
            begin
                fc_en <= 1'b1;
                weight_base_addr <= fc1_weight_base_addr;
                bias_base_addr <= fc1_bias_base_addr;
                data_base_addr <= fc1_data_base_addr;
                result_base_addr <= fc1_result_base_addr;
                rownum <= 10'd16;
                fc_columnnum <= 15'd25088;
                state <= S_FC1;
            end
            5'b10011:  // 19 fc2
            begin
                fc_en <= 1'b1;
                weight_base_addr <= fc2_weight_base_addr;
                bias_base_addr <= fc2_bias_base_addr;
                data_base_addr <= fc2_data_base_addr;
                result_base_addr <= fc2_result_base_addr;
                rownum <= 10'd128;
                fc_columnnum <= 15'd4096;
                state <= S_FC2;
            end
            5'b10100:  // 20 fc3
            begin
                fc_en <= 1'b1;
                weight_base_addr <= fc3_weight_base_addr;
                bias_base_addr <= fc3_bias_base_addr;
                data_base_addr <= fc3_data_base_addr;
                result_base_addr <= fc3_result_base_addr;
                rownum <= 10'd125;
                fc_columnnum <= 15'd4096;
                state <= S_FC3;
            end
```

```
                    default:
                    begin
                        conv_en <= 1'b0;
                        pool_en <= 1'b0;
                        fc_en <= 1'b0;
                        state <= S_IDLE;
                    end
                endcase
            end
            else
            begin
                vgg16_finish <= 1'b0;
                state <= S_IDLE;
            end
        end
        S_CONV1_1:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV1_2:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_POOL1:
        begin
            if(pool_finish == 1'b1)
            begin
                pool_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV2_1:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV2_2:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
```

```
                S_POOL2:
                begin
                    if(pool_finish == 1'b1)
                    begin
                        pool_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_CONV3_1:
                begin
                    if(conv_finish == 1'b1)
                    begin
                        conv_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_CONV3_2:
                begin
                    if(conv_finish == 1'b1)
                    begin
                        conv_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_CONV3_3:
                begin
                    if(conv_finish == 1'b1)
                    begin
                        conv_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_POOL3:
                begin
                    if(pool_finish == 1'b1)
                    begin
                        pool_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_CONV4_1:
                begin
                    if(conv_finish == 1'b1)
                    begin
                        conv_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_CONV4_2:
                begin
                    if(conv_finish == 1'b1)
                    begin
                        conv_en <= 1'b0;
                        vgg16_finish <= 1'b1;
```

```
                state <= S_IDLE;
            end
        end
        S_CONV4_3:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_POOL4:
        begin
            if(pool_finish == 1'b1)
            begin
                pool_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV5_1:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV5_2:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_CONV5_3:
        begin
            if(conv_finish == 1'b1)
            begin
                conv_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_POOL5:
        begin
            if(pool_finish == 1'b1)
            begin
                pool_en <= 1'b0;
                vgg16_finish <= 1'b1;
                state <= S_IDLE;
            end
        end
        S_FC1:
        begin
            if(fc_finish == 1'b1)
```

```
                    begin
                        fc_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_FC2:
                begin
                    if(fc_finish == 1'b1)
                    begin
                        fc_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                S_FC3:
                begin
                    if(fc_finish == 1'b1)
                    begin
                        fc_en <= 1'b0;
                        vgg16_finish <= 1'b1;
                        state <= S_IDLE;
                    end
                end
                default:
                begin
                    state <= S_IDLE;
                    conv_en <= 1'b0;
                    pool_en <= 1'b0;
                    fc_en <= 1'b0;
                end
            endcase
        end
    end
```

6、对最后实现的 vgg16_top 模块进行仿真并验证（只是随便找了一些数据）。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
        // vgg16 calculate
        if(slv_reg0[0] == 1'b1 && slv_reg0[1] == 1'b0 && slv_reg0[2] == 1'b0)
        begin
            if(i == 0)
            begin
                rst <= 1'b1;
                layer <= slv_reg3[4:0];
                addbefore <= slv_reg2[0];
                base_addr <= 20'b0;
                num <= 32'b0;
                slv_reg1 <= 32'b0;
                i <= i + 1;
            end
            else if(i == 1)
            begin
                rst <= 1'b0;
                vgg16_start <= 1'b1;
                i <= i + 1;
```

```
            end
            else
            begin
                if(i < 4)
                begin
                    i <= i + 1;
                end
                else if(i == 4)
                begin
                    vgg16_start <= 1'b0;
                    i <= i + 1;
                end
                else
                begin
                    if(vgg16_finish == 1'b1)
                    begin
                        slv_reg1[0] <= vgg16_finish;
                        slv_reg1[1] <= 1'b0;
                        slv_reg1[0] <= 1'b0;
                        i <= 0;
                    end
                end
            end
        end
        // loaddma
        else if(slv_reg0[0] == 1'b0 && slv_reg0[1] == 1'b1 && slv_reg0[2] == 1'b0)
        begin
            if(i == 0)
            begin
                rst <= 1'b1;
                layer <= 5'b0;
                addbefore <= 1'b0;
                base_addr <= slv_reg4[19:0];
                num <= slv_reg5;
                slv_reg1 <= 32'b0;
                i <= i + 1;
            end
            else if(i == 1)
            begin
                rst <= 1'b0;
                loaddma_start <= 1'b1;
                i <= i + 1;
            end
            else
            begin
                if(i < 4)
                begin
                    i <= i + 1;
                end
                else if(i == 4)
                begin
                    loaddma_start <= 1'b0;
                    i <= i + 1;
                end
                else
                begin
                    if(loaddma_finish == 1'b1)
                    begin
                        slv_reg1[0] <= 1'b0;
                        slv_reg1[1] <= loaddma_finish;
```

```
                        slv_reg1[0] <= 1'b0;
                        i <= 0;
                    end
                end
            end
        end
        // storedma
        else if(slv_reg0[0] == 1'b0 && slv_reg0[1] == 1'b0 && slv_reg0[2] == 1'b1)
        begin
            if(i == 0)
            begin
                rst <= 1'b1;
                layer <= 5'b0;
                addbefore <= 1'b0;
                base_addr <= slv_reg4[19:0];
                num <= slv_reg5;
                slv_reg1 <= 32'b0;
                i <= i + 1;
            end
            else if(i == 1)
            begin
                rst <= 1'b0;
                storedma_start <= 1'b1;
                i <= i + 1;
            end
            else
            begin
                if(i < 4)
                begin
                    i <= i + 1;
                end
                else if(i == 4)
                begin
                    storedma_start <= 1'b0;
                    i <= i + 1;
                end
                else
                begin
                    if(storedma_finish == 1'b1)
                    begin
                        slv_reg1[0] <= 1'b0;
                        slv_reg1[1] <= storedma_finish;
                        slv_reg1[0] <= 1'b0;
                        i <= 0;
                    end
                end
            end
        end
    end
```

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

在 Implementation 时和实验二一样遇到了一个 *critical warning：[Timing 38-282] The design failed to meet the timing requirements.* 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：（只选一个卷积层、一个池化层和一个全连接层为例）从 SD 卡中读入文件并每次选择参数已略去。

```
for(layercount = 0; layercount < 21; layercount++){
        // conv1_1
        if(layercount == 0){
                // open files in sdk and read
                for(int j = 0; j < CONV1_1_CALCOUNT; j++){
                        u32 r;
                        // load data
                        for(int i = 0; i < CONV1_1_DATALOADCOUNT; i++){
                                // load data to dma bram
                                loaddma(conv1_1_data_base_addr+i*DMANUM,
                                        DMANUM);
                                while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                        if(r == 0x00000002){
                                                break;
                                        }
                                }
                        }
                        // load weight to dma bram
                        // load weight
                        loaddma(conv1_1_weight_base_addr, 3*3*3*4);
                        while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000002){
                                        break;
                                }
                        }
                        // load bias to dma bram
                        // load bias
                        loaddma(conv1_1_bias_base_addr, 4);
                        while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000002){
                                        break;
                                }
                        }
                        // vgg16 calculate
                        vgg16_calculate(layercount, 0);
                        while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000001){
                                        break;
                                }
                        }
                        // store result
                        for(int i = 0; i < CONV1_1_RESULTSTORECOUNT; i++){
                                storedma(conv1_1_result_base_addr+i*DMANUM,DMANUM);
                                while(1){
```

```
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4)
                            if(r == 0x00000004){
                                break;
                            }
                        }
                        // store bram to dma
                    }
                }
            }
            // pool1
            if(layercount == 2){
                // open files in sdk and read
                for(int j = 0; j < POOL1_CALCOUNT; j++){
                    u32 r;
                    // load data
                    for(int i = 0; i < POOL1_DATALOADCOUNT; i++){
                        // load data to dma bram
                        loaddma(pool1_data_base_addr+i*DMANUM, DMANUM);
                        while(1){
                            r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                            if(r == 0x00000002){
                                break;
                            }
                        }
                    }
                    // vgg16 calculate
                    vgg16_calculate(layercount, 0);
                    while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                        if(r == 0x00000001){
                            break;
                        }
                    }
                    // store result
                    for(int i = 0; i < POOL1_RESULTSTORECOUNT; i++){
                        storedma(pool1_result_base_addr+i*DMANUM,DMANUM);
                        while(1){
                            r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                            if(r == 0x00000004){
                                break;
                            }
                        }
                        // store bram to dma
                    }
                }
            }
            // fc3
            else if(layercount == 20){
                // open files in sdk and read
                for(int j = 0; j < FC3_CALCOUNT; j++){
                    u32 r;
                    // load data
                    // load data to dma bram
                    loaddma(fc3_data_base_addr, 4096);
                    while(1){
                        r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                        if(r == 0x00000002){
                            break;
                        }
                    }
```

```
                        // load weight
                        for(int i = 0; i < FC3_WEIGHTLOADCOUNT; i++){
                                // load weight to dma bram
                                if(i == FC3_WEIGHTLOADCOUNT - 1){
                                        loaddma(fc3_weight_base_addr+i*8192, 4096);
                                }
                                else{
                                        loaddma(fc3_weight_base_addr+i*8192, 4096*2);
                                }
                                while(1){
                                    r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                        if(r == 0x00000002){
                                                break;
                                        }
                                }
                        }
                        // load bias to dma bram
                        // load bias
                        loaddma(fc3_bias_base_addr, 125);
                        while(1){
                             r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000002){
                                        break;
                                }
                        }
                        // vgg16 calculate
                        vgg16_calculate(layercount, 0);
                        while(1){
                            r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000001){
                                        break;
                                }
                        }
                        // store result
                        storedma(fc3_result_base_addr,125);
                        while(1){
                            r = VGG16_mReadReg(XPAR_VGG16_0_S00_AXI_BASEADDR, 4);
                                if(r == 0x00000004){
                                        break;
                                }
                        }
                        // store bram to dma
                }
        }
}
```

9、关于仿真验证过程中的调试

由于时间关系，没有进行调试。

10、数据预处理阶段的代码(octave 定点化代码)，源代码 fixed.m 已和工程一同提交。

```
load temp.mat
input = temp;
s = size(temp)
prec_f = 6;
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);
size(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
            a = dec2bin(data_out(i, j), 7);
            col = (j-1)*9 + 1;
            if(a(1) == '1')
                data_out_bin(i,col) = '1';
            else
                data_out_bin(i,col) = '0';
            end
            data_out_bin(i,col+1:col+7) = a;
            data_out_bin(i,col+8) = ' ';
        end
end

size(data_out_bin)

save result data_out_bin
```

七、实验结果

由于时间关系，最后并没有进行 SDK 端代码的 debug 工作以及上板实测的工作。

八、分析与讨论

1、对于 VGG16 网络本次依然采用了和 LeNet 网络一样的单周期设计，效率非常低，必须要进行流水化处理；

2、小的 Block RAM 感觉有点浪费，主要是我不知道怎么修改和 axi bram controller 接口连接的 Block RAM 的大小，不然应该可以不需要小的这块 Block RAM；

3、其他优化都没有尝试，不过我认为主要的优化方向就是尽量减少 PS 和 PL 端的交互次数以及 PL 端从 Block RAM 中读取数据的次数；