智能计算体系结构课程lab4实验报告

作者：北京航空航天大学计算机学院 陈胤佳

学号：19373383

一、实验目的与要求：

本次实验目的为，在Lab3移植Linux至ZYNQ7020开发板、并测试开发环境的基础上，进一步挂载BRAM，使得ZYNQ上的Linux操作系统能够对其进行读写操作；设计软硬件交互逻辑、并实现矩阵乘法Matmul接口。

二、实验环境：

IDE：Vivado 2019.2 / Vitis 2019.2；

Environment：ZYNQ 7020开发板及其配件；

Environment：ZYNQ 7020上的Linux系统；

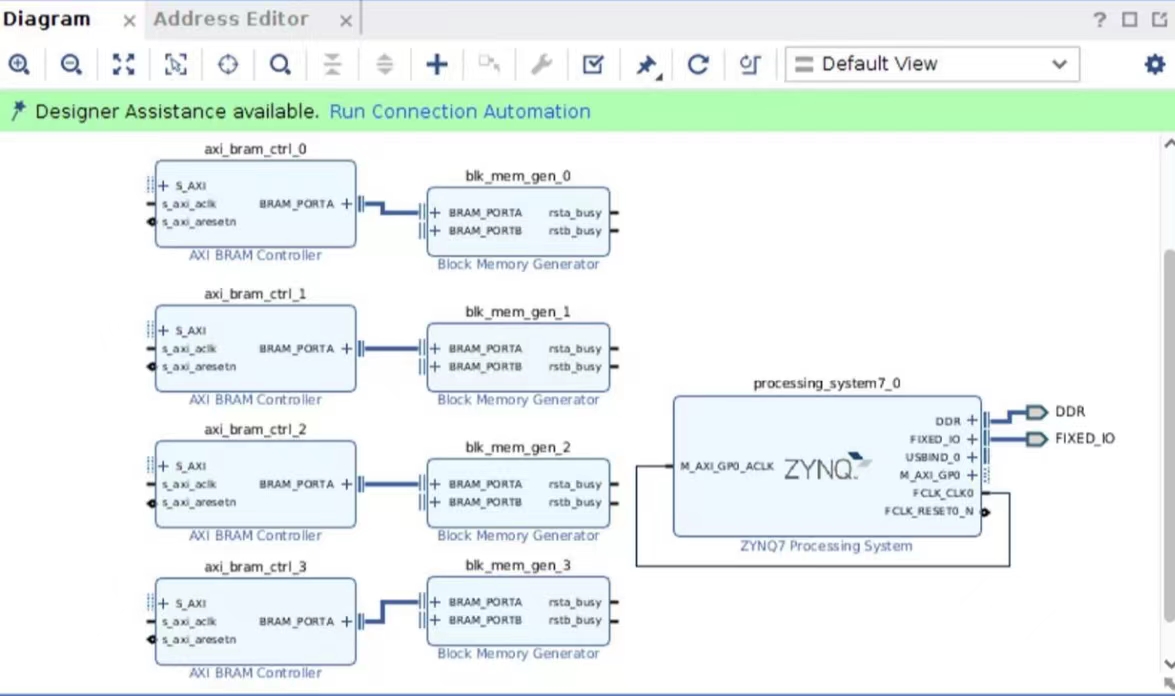
OS：Windows 10 Professional

三、实验内容与步骤

1. 制作挂载BRAM的bit流文件：

1. Vivado工程添加BRAM：

采用课程组提供的linux\_zynq\_7020\_vivado\_2019.2工程目录及其中文件，构建Vivado工程。首先使用Vivado加载目录下的.xpr文件载入工程；点击Open Block Design，添加block memory generator，将BRAM配置为双口bram；再次点击Open Block Design，添加axi bram controller，配置bram interface为1。将上述添加的两个单元成组复制3次，并进行如图所示的连接：



最后，点击run connection automation自动连线，规划地址信息，BRAM添加完毕。

1. 生成bit流并下载至开发板：

点击Generate Bitstream生成bit流，打开硬件管理器连接设备、并将bit流下载至开发板。

2. 在ZYNQ的linux中测试BRAM读写

本步骤核心代码位于bram.py中，可以将其移动至root/目录下上板测试。下对该文件进行解读。

1. BramConfig类：

用于配置BRAM信息，包括块名称、起始地址、大小、各偏移量名称及其对应数值。其中包含方法\_construct\_block\_info(address, size, \*\*offset)：



该方法返回字典格式数据，用于构造block信息，在初始化BramConfig，即BRAM地址分配信息时需要调用，从而构建block\_info。block\_info构建过程如下：



1. BRAM类：

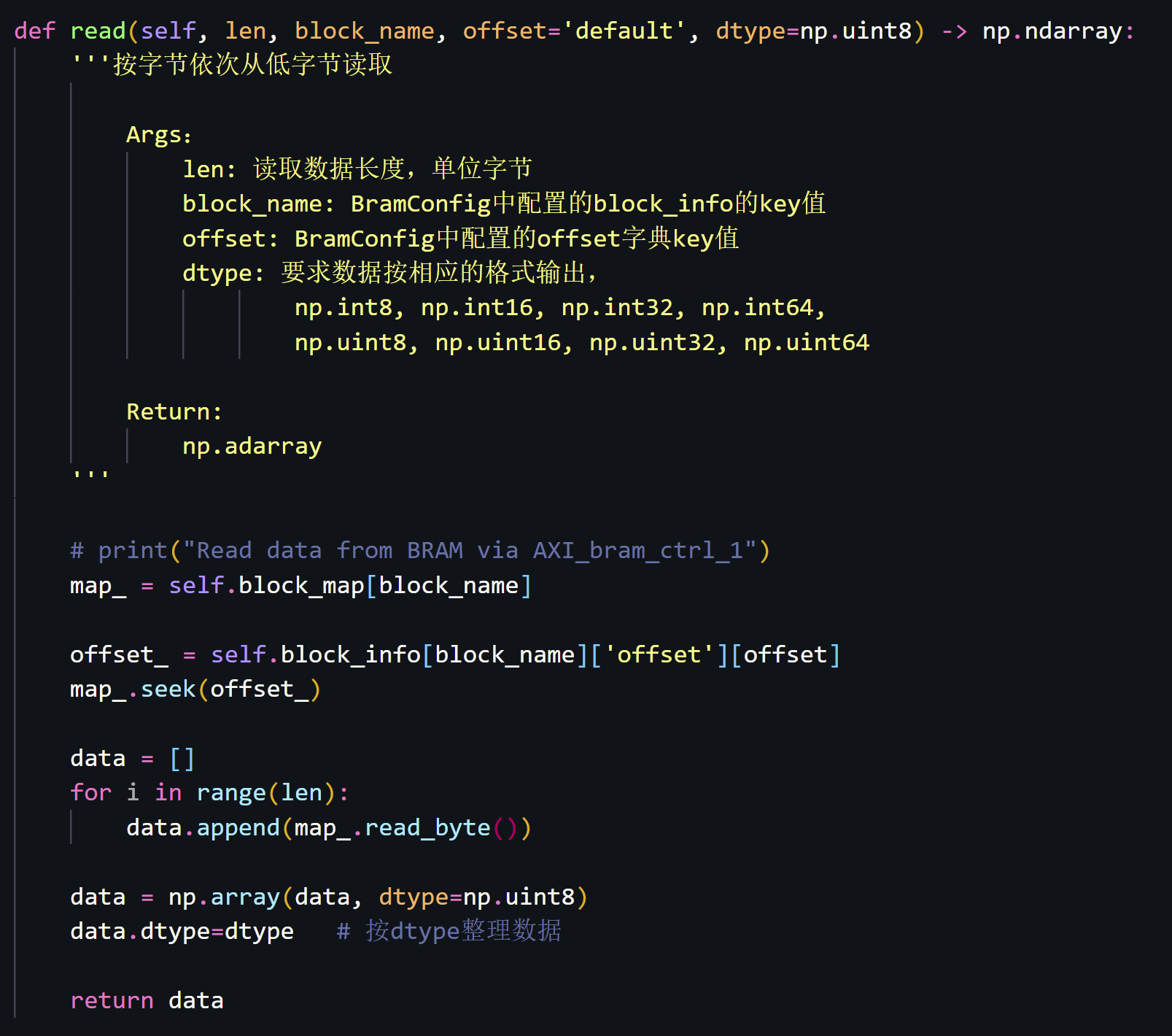
主要实现对挂载BRAM的读写。

1. write方法：



可以将data（np.ndarry或bytes格式数据）写入block\_name块对应offset的位置。

1. read方法：

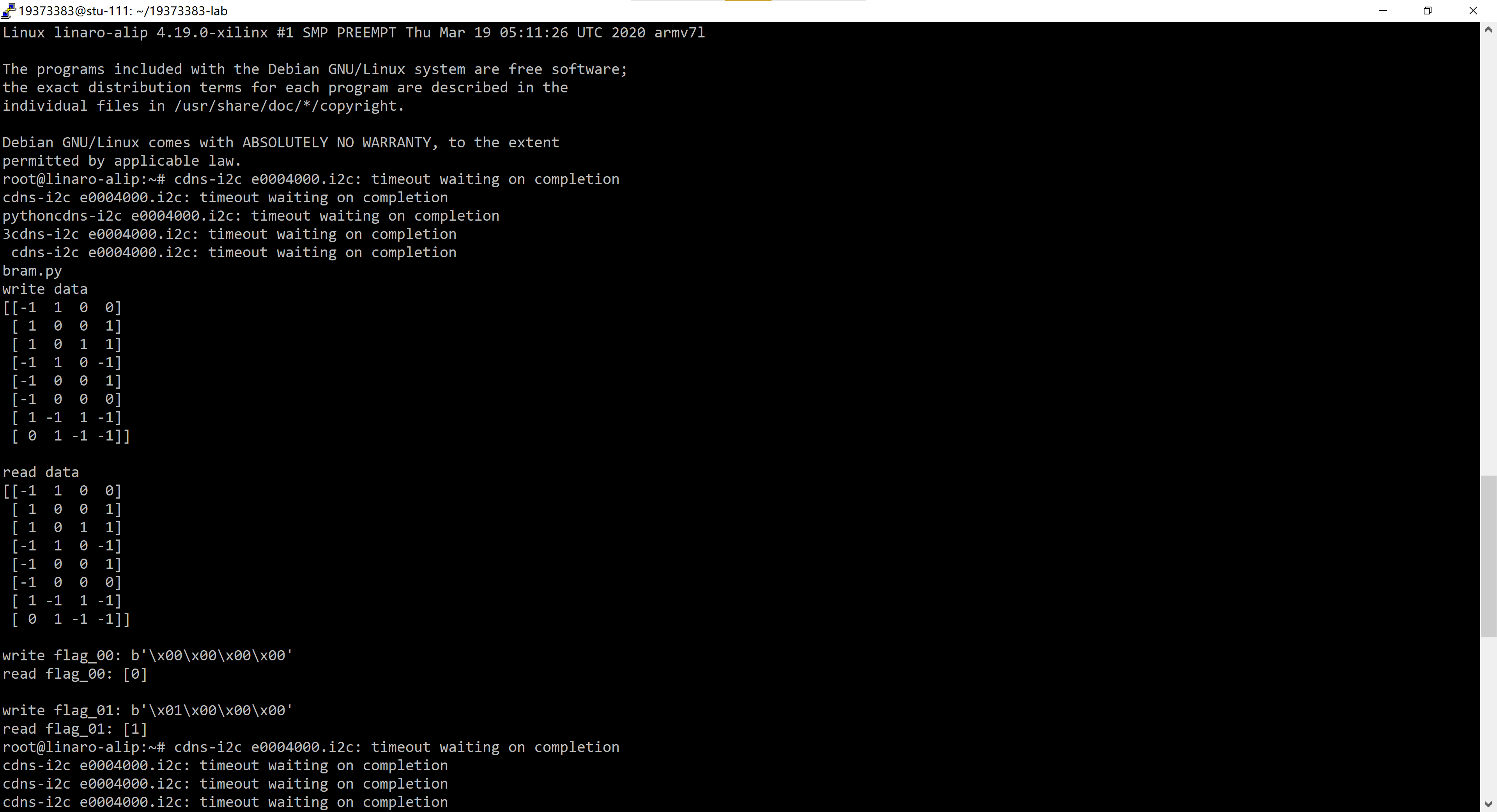


可以将block\_name块中offset对应长度为len的数据读出，返回数据类型为np.ndarray，dtype为指定的其中的数据类型。

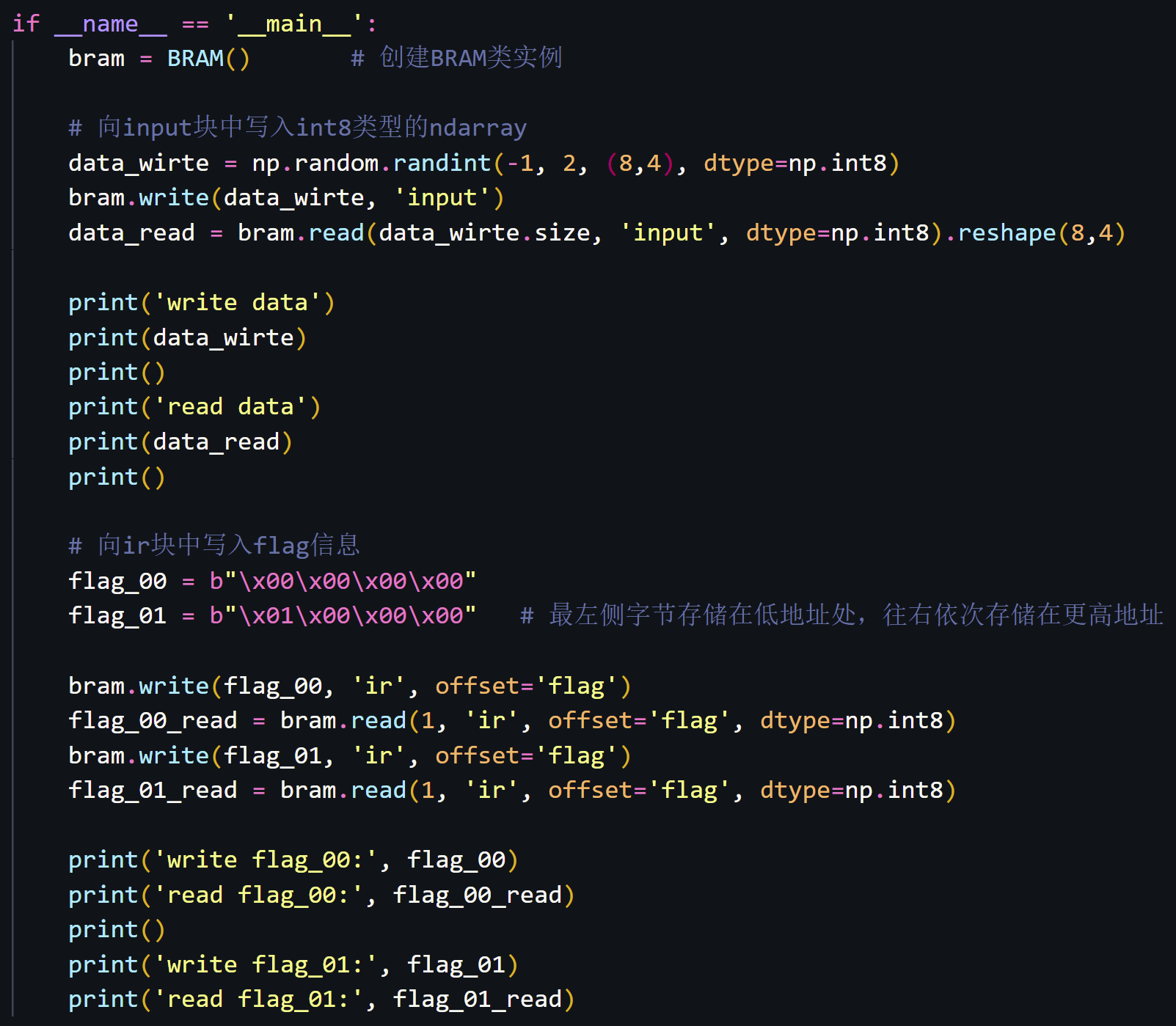
1. 测试结果：

在putty打开的端口中输入python3 bram.py，

结果如下：



解释：源码如下。



主函数分别测试了向BRAM块中写ndarray、byte格式数据的读写。首先，程序声明定义了一个8行4列的随机数矩阵，其中元素的取值范围为-1到2，数据类型为np.int8；随后将该矩阵写入BRAM的input块的default\_offset对应的位置中。接下来对其进行读取（注意，由于存储的shape格式丢失，读出后需要进行reshape），并打印结果，观察是否一致。Byte情况同理。

3. 矩阵乘法软硬件交互设计

1. 设计方案

ZYNQ板子分为ARM侧与FPGA侧，需要两侧的交互才能够完成计算，交互流程如下：

1. ARM端：将大小为、的矩阵分别存入input和weight的BRAM中。
2. ARM端：按照指令格式构造指令，存入ir块对应的instr偏移中。
3. ARM端：写入flag\_01信号，即将1写入ir块地址对应的flag偏移中。
4. FPGA端：从对应BRAM块中取出数据（input、weight、指令），执行矩阵乘法，并将结果写入output对应的BRAM块中。
5. FPGA端：写入flag\_00信号，即0写入ir块地址对应的flag偏移位置中。
6. ARM端：读结果，一次矩阵乘法结束。
7. Matmul.py文件说明及补全

class Matmul(object):

    '''矩阵乘法

        Args: uint8, (m, n)

        Args: int8, (n, p)

    '''

    def \_\_init\_\_(self):

        self.systolic\_size = 4 # 脉动阵列大小

        self.bram = BRAM()

    def \_\_call\_\_(self, input: np.uint8, weight: np.int8):

        self.send\_data(input, 'input')

        self.send\_data(weight, 'weight')

        output\_arr = self.recv\_output((None, None))

        return output\_arr

    def send\_data(self, data, block\_name, offset='default'):

        '''写入input或weight至bram

假设两个矩阵分别是(m,n) x (n,p), m和p的维度

需要补全至self.systolic\_size的倍数，

            并且写入时需要按照补零的方向写入，例如：

                1. 矩阵(m, n)是m补零，则m个m个写入BRAM中（行方向补零，列方向写入）

                2. 矩阵(n, p)是p补零，则p个p个写入BRAM中。（列方向补零，行方向写入）

            Args:

                data: 要写入的数据

                block\_name: input, weight

                offset: 偏移地址名称，默认为default

        '''

        self.bram.write(data, block\_name, offset)

    def send\_instr(self, m, p, n):

        '''构建并发送指令

            两个矩阵shape分别为(m,n) x (n,p)

        '''

        m\_hex = m << 48

        p\_hex = p << 32

        n\_hex = n << 16

        instr = (m\_hex + p\_hex + n\_hex).to\_bytes(8, byteorder='big')

        self.bram.write(instr, 'ir', offset='instr')

    def send\_flag(self):

        '''发送flag=1信号'''

        # 向ir块中写入flag信息

        flag\_01 = b"\x01\x00\x00\x00"   # 最左侧字节存储在低址处，往右依次存储在更高地址

        self.bram.write(flag\_01, 'ir', offset='flag')

    def recv\_output(self, output\_shape: tuple):

        '''接收结果

            Args:

                output\_shape: 输出的shape，类型tuple

            Return:

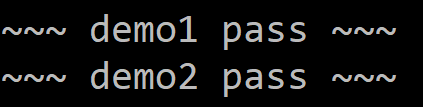
                output\_arr: shape为output\_shape的np.ndarray

        '''

        output\_arr = self.bram.read(output\_shape[0] \* output\_shape[1], 'output', dtype=np.int8).reshape(output\_shape)

        return output\_arr

1. 测试结果：
2. 安装tmux
3. 开启一个新的tmux窗口并运行pl\_simulate.py（用root权限运行）
4. 脱离tmux终端
5. 运行Matmul.py测试矩阵乘法，结果如下：



四、实验结果

环境配置成功，两文件测试结果详见实验步骤。

五、分析与讨论

本次实验熟悉了板子上BRAM的软硬件交互流程，并通过已给出的测试文件，理解了BRAM的读写方式以及软硬件交互的基本流程。最后，配置环境真的是一个艰难困苦的过程，本次实验的系统装配以及环境搭建令我留下了难忘的印象。