

Lab 4：设计软硬件交互逻辑

1 实验目标

- 1) 挂载 BRAM，使得 ZYNQ 上的 Linux 操作系统能够对其进行读写操作
- 2) 设计软硬件交互逻辑
- 3) 实现矩阵乘法 Matmul 接口

2 实验环境

- 1) Vivado 2019.2 / Vitis 2019.2
- 2) ZYNQ 7020 开发板及其配件
- 3) ZYNQ 上的 Linux 系统
- 4) 本实验过程中使用到的数据文件可在北航盘下载
- 5) 注意：本实验指导书中给出的步骤仅为示意步骤作为参考，每人遇到的情况可能有差异，如果遇到问题可根据实际情况进行探索，或向助教寻求帮助。

3 实验要求

- 1) 撰写实验报告并回答 4.1.7 小节中提出的问题。实验报告命名：学号+姓名+实验四。（实验报告撰写细节可参考“实验报告撰写格式”）`
- 2) 将实验报告按时交至课程中心作业处。

4 实验内容

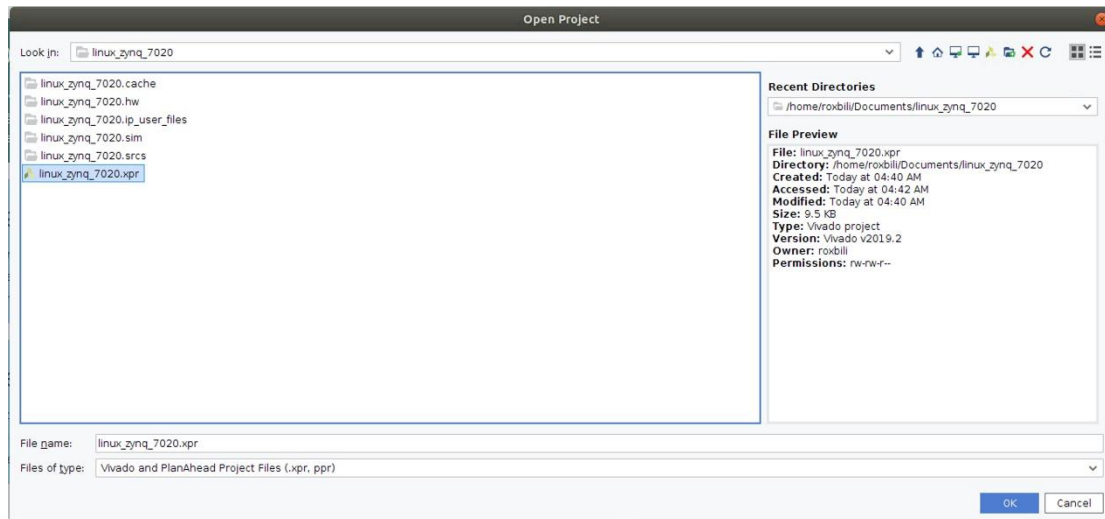
4.1 制作挂载 BRAM 的 bit 流文件

实验 Vivado 工程为 linux_zynq_7020，解压 linux_zynq_7020_vivado_2019.2.zip 安装包后即可看到。

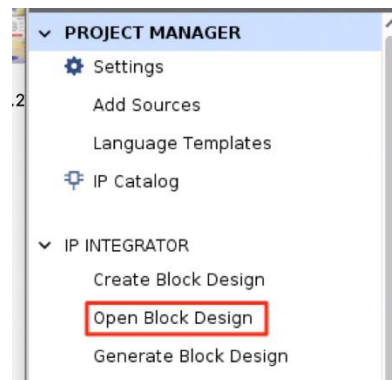
4.1.1 Vivado 工程添加 BRAM

由于本课程中 PS-PL 侧数据交互均采用 BRAM 存取实现，因此需要将 BRAM 挂载至 PS-PL 两侧。本实验中，仅测试 PS 侧对 BRAM 的读写操作，因此 BRAM 暂时只挂载在 PS 侧。

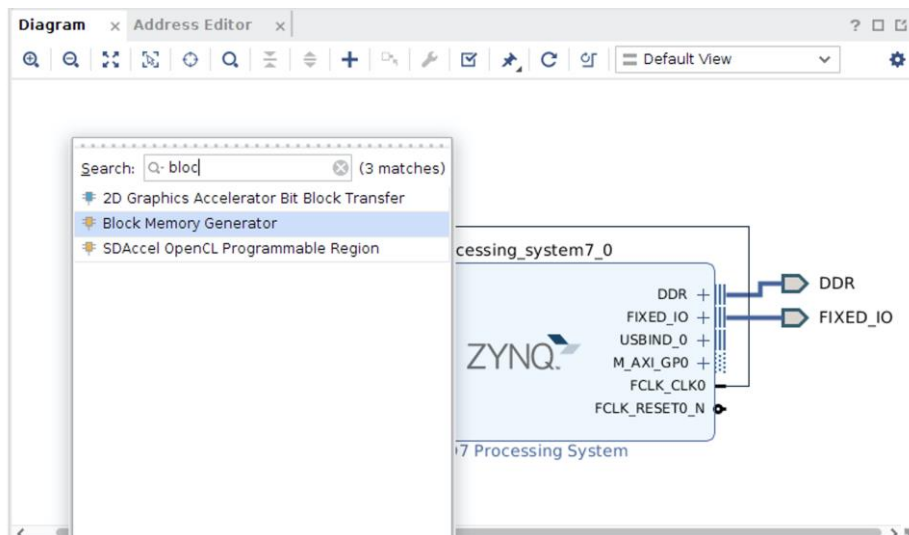
- 1) 使用 Vivado 打开工程文件



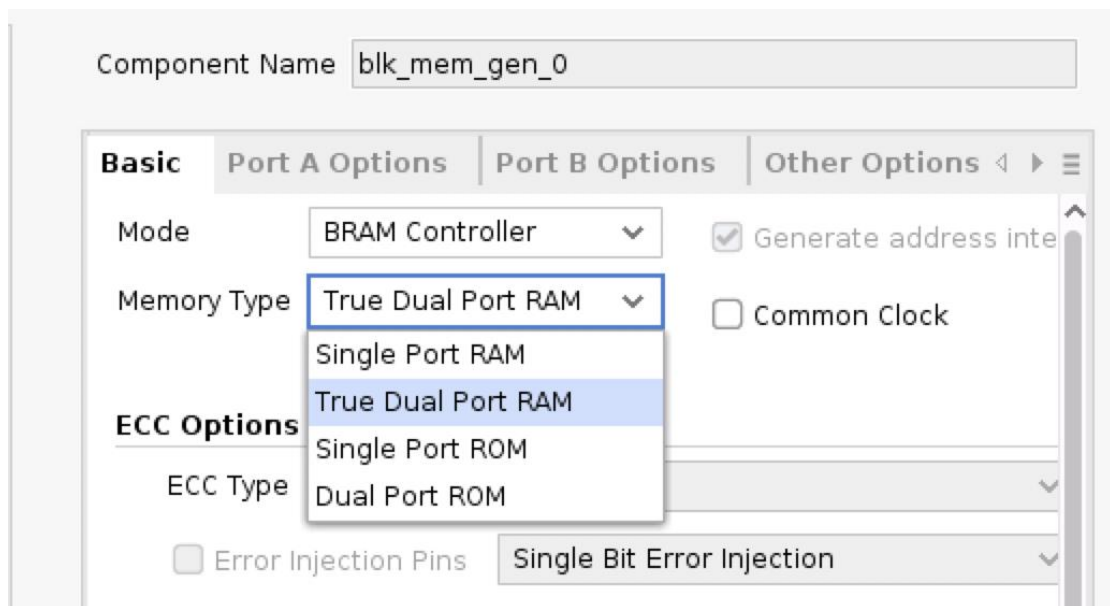
2) 打开工程后点击左侧 Open Block Design



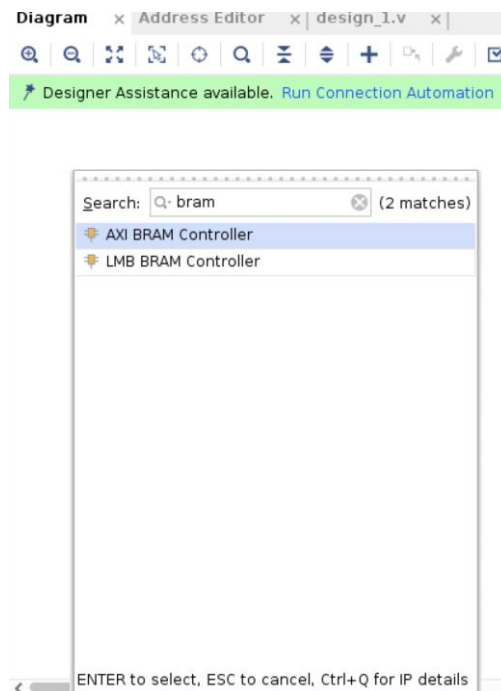
3) 点击+号，搜索 block，添加 block memory generator



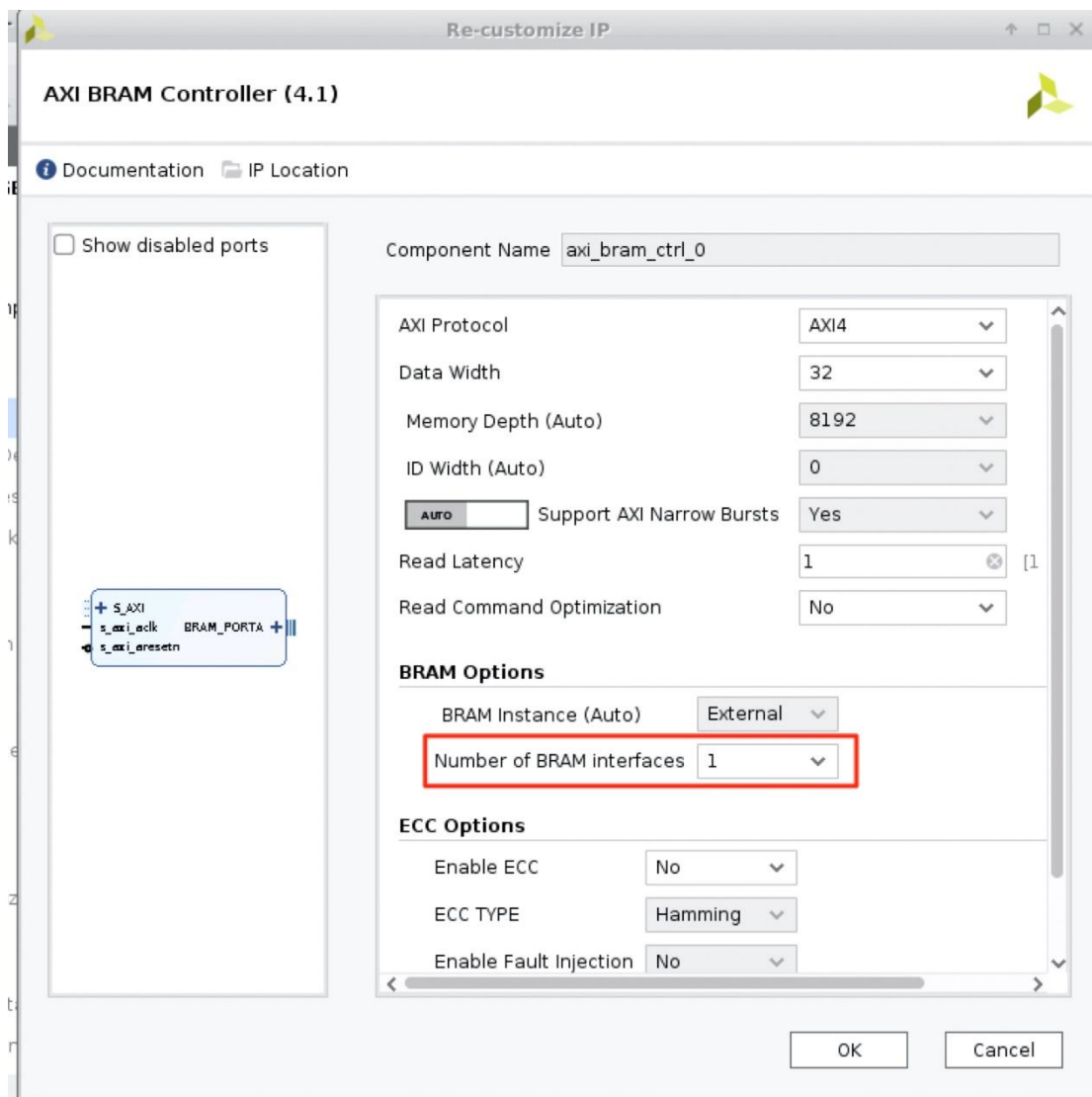
双击 BRAM，打开配置菜单，选择双口 bram。本实验不涉及 PL，因此仅对一个端口进行连线。



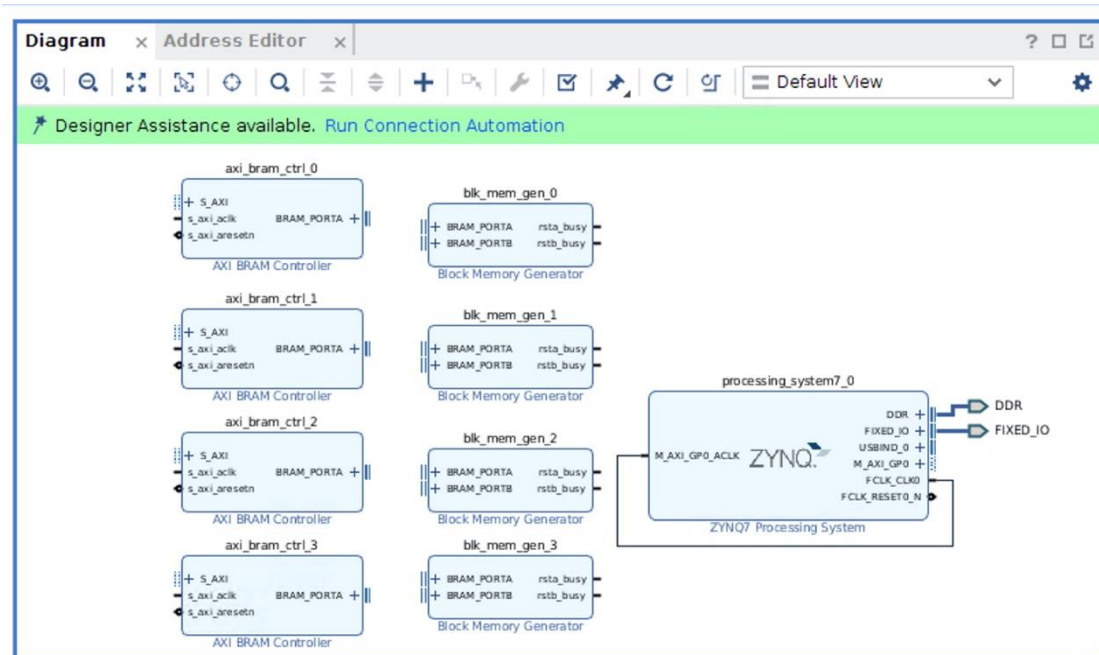
再次点击+号，搜索 bram，添加 axi bram controller



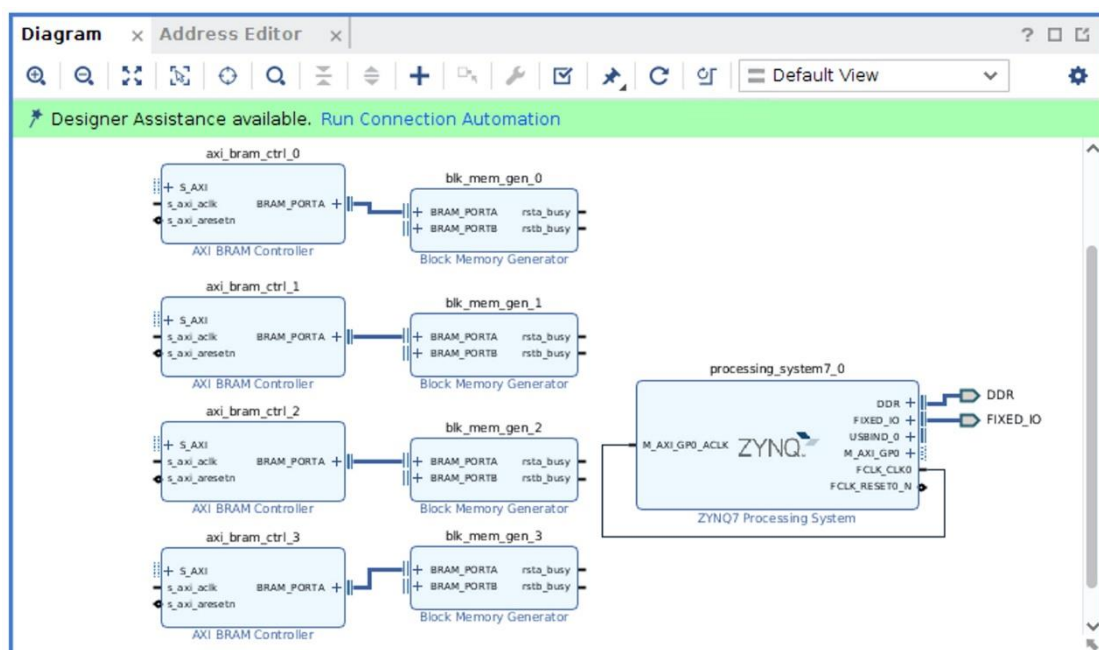
双击新添加的 axi bram controller，配置 bram interface 为 1。



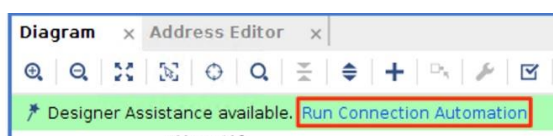
- 4) 选中 axi bram controller 和 block memory generator，复制 3 次，即最终将有 4 个 axi bram controller 和 4 个 block memory generator。

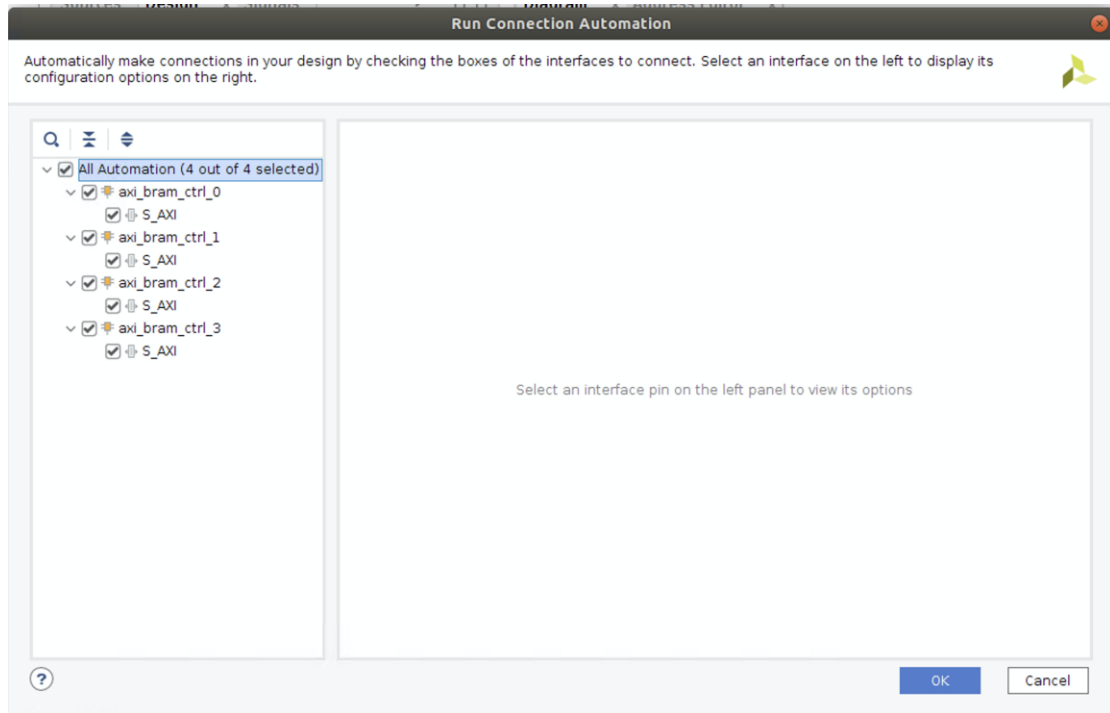


5) 将 axi_bram_ctrl 的 BRAM_PORTA 连接到 blk_mem_gen 的 BRAM_PORTA 上。(4 个都做同样的操作)

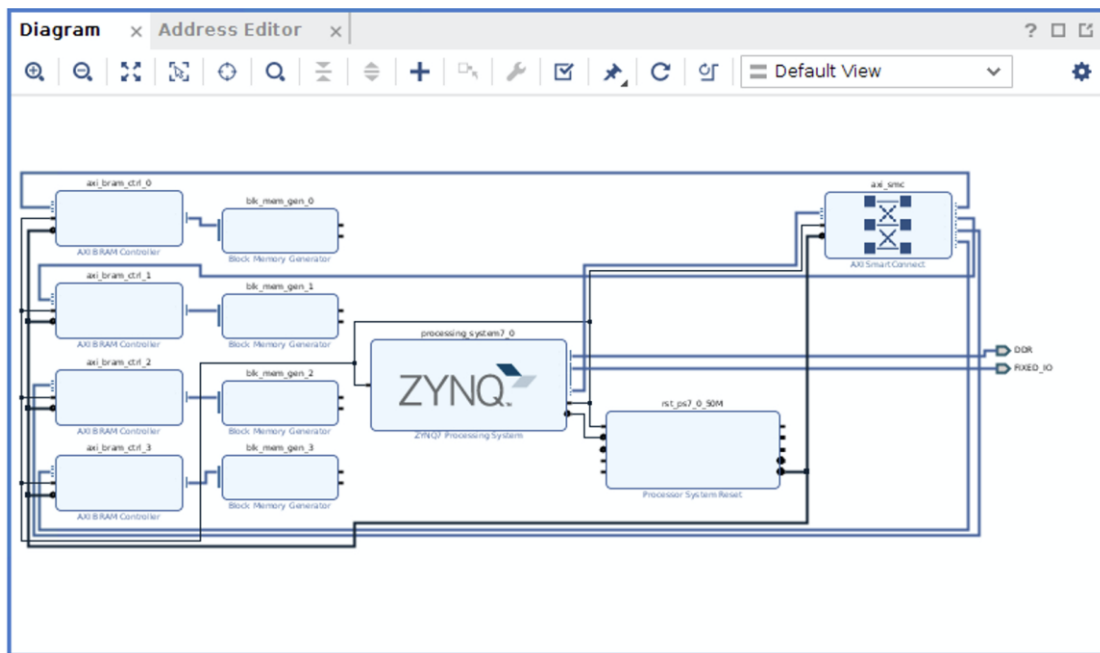


6) 点击 run connection automation 自动连线，左侧面板全选后点击 OK



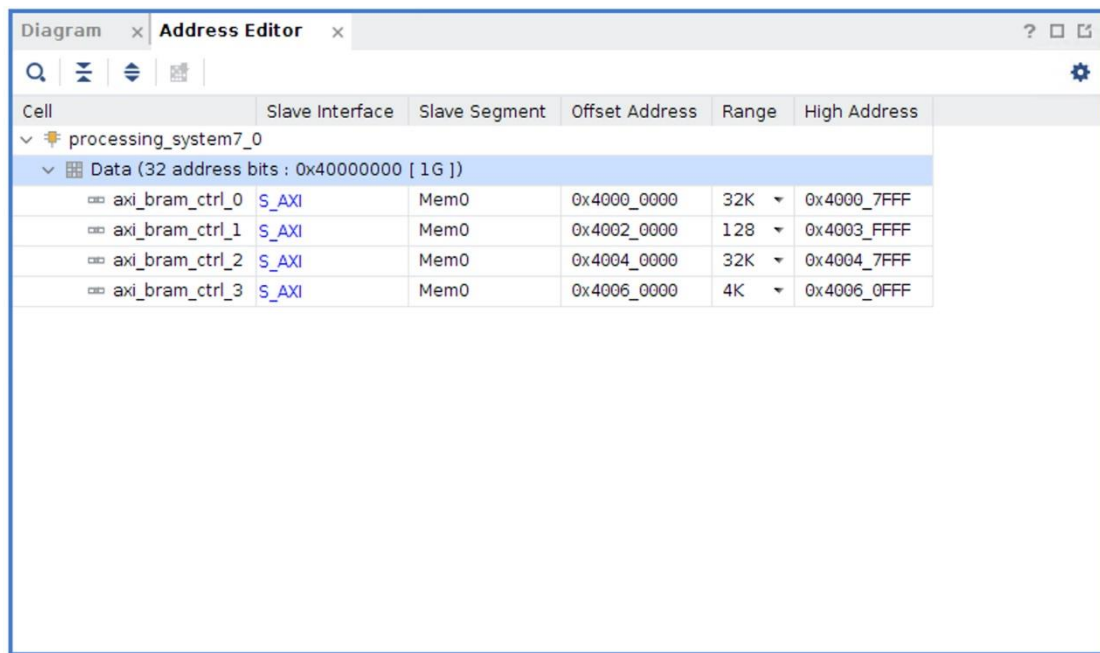


最终连接效果如图所示。



7) 规划地址信息

点击 Address Editor，填写 axi_bram_ctrl_0-3 的 Offset Address 和 Range 字段如下图所示：



Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	32K	0x4000_7FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4002_0000	128	0x4003_FFFF
axi_bram_ctrl_2	S_AXI	Mem0	0x4004_0000	32K	0x4004_7FFF
axi_bram_ctrl_3	S_AXI	Mem0	0x4006_0000	4K	0x4006_0FFF

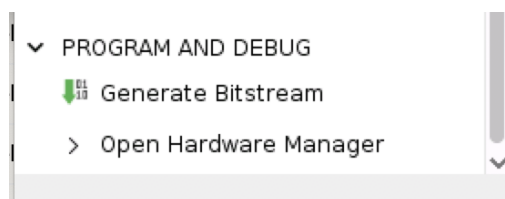
4.1.2 生成 bit 流并下载至开发板

1) 点击左侧面板生成 bit 流

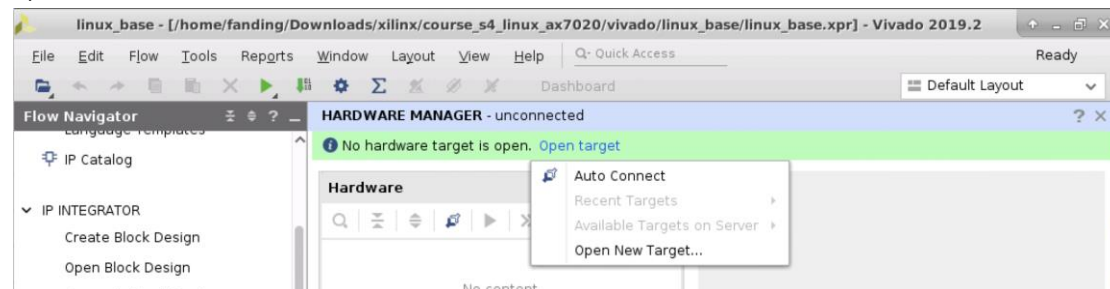


使用默认选项点击 **ok** 开始生成即可。

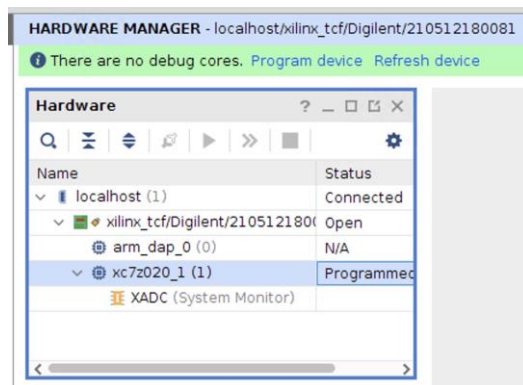
2) 在左侧面板打开硬件管理器



3) 连接目标设备

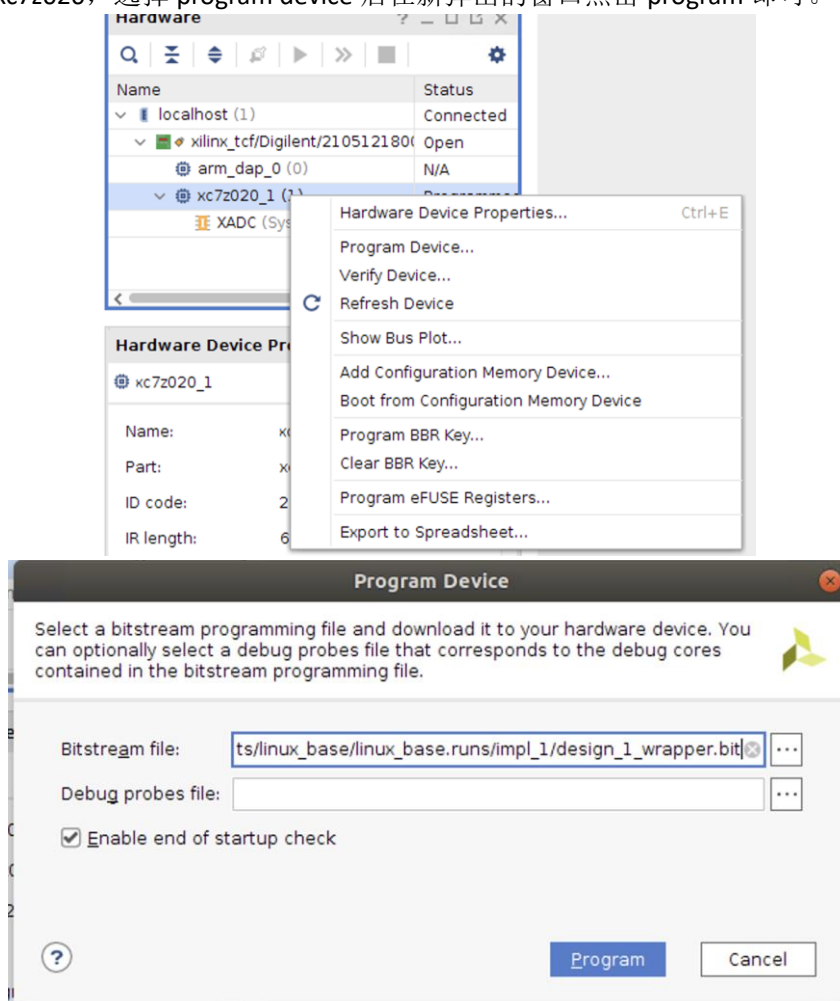


(若连接报错, 请在虚拟机设置里 USB 兼容接口改为 3.0 或以上)
连接成功后如下图所示:



4) 下载 bit 流至开发板

右键单击 xc7z020，选择 program device 后在新弹出的窗口点击 program 即可。



bit 流下载成功后即可开始测试。

（需要注意，每次重启 ZYNQ 板子后都需要重新写入 bit 流）

4.2 在 ZYNQ 上的 Linux 系统中测试 BRAM 读写

本实验提供与底层 BRAM 读写的技术代码，见 `bram.py`，其中实现两个类，分别是 `BramConfig` 和 `BRAM`，下面分别进行介绍。

4.2.1 BramConfig

BramConfig 类用于配置 BRAM 相关信息，包括块名称、块起始地址、块大小、块偏移量名称和块内偏移数值。在 BRAM 类中将会使用到这些信息。

block_info 用于存储块信息，块信息使用函数_construct_block_info 构造，该函数输入及返回可见函数注释部分：

```
def _construct_block_info(address, size, **offset) -> dict:
    '''构造block信息

    Args:
        name: 块名称
        address: 块起始地址
        size: 块大小
        offset: 偏移量, 字典

    Return:
        返回字典, 包含address, size, offset字段。
        其中offset是一个字典, 表示各块内偏移的用途
    '''
    info = {
        'address': address,
        'size': size,
        'offset': offset
    }
    return info
```

具体函数使用方法可参考以下截图，最后所有块信息将存储在 block_info 字典变量中。截图中共分为 4 块，块名称(block_name)分别为 input、weight、output、ir，含义如下：

- input: 存储输入，偏移量默认从块头部开始存储数据，即为 default
- weight: 存储权重，偏移量默认从块头部开始存储数据，即为 default
- output: 存储输出，偏移量默认从块头部开始存储数据，即为 default
- ir: 存储标记和指令，对应偏移量名称为 flag、instr。flag 用于 PS、PL 交互，从块头部开始存储，instr 从偏移 0x10 字节的位置开始存储。

```
block_info = {}
# 构建新逻辑块参考以下写法
# 若块内偏移量无特殊含义，则约定key为default，值为0，可根据实际需求修改
block_info['input'] = _construct_block_info(
    address=0x40000000, size=32*1024,
    **{'default': 0x0}
)
block_info['weight'] = _construct_block_info(
    address=0x40020000, size=128*1024,
    **{'default': 0x0}
)
block_info['output'] = _construct_block_info(
    address=0x40040000, size=32*1024,
    **{'default': 0x0}
)
block_info['ir'] = _construct_block_info(
    address=0x40060000, size=4*1024,
    **{'flag': 0x0, 'instr': 0x10}
)
```

4.2.2 BRAM

BRAM 类实现对挂载的 BRAM 进行读写，分别为 write 和 read 方法。

1. write

write 函数输入参数可见函数注释。其中，data 为输入的数据，可输入 np.ndarray 或者 bytes 类型的数据；block_name 即块名称；offset 即偏移量名称。

```
def write(self, data, block_name: str, offset='default'):
    '''写入数据
        由于数据位宽32bit, 因此最好以4的倍数Byte写入(还不知道以1Byte单位写进去会有什么效果)

    Args:
        data: 输入的数据
        block_name: BramConfig中配置的block_info的key值
        offset: BramConfig中配置的offset字典key值
    '''
    map_ = self.block_map[block_name]

    # print("Data: \n%s" % data)
    offset_ = self.block_info[block_name]['offset'][offset]
    map_.seek(offset_)

    if isinstance(data, np.ndarray):
        data = data.reshape(-1)
    map_.write(data)
```

2. read

read 函数输入参数可见函数注释。其中, len 表示读取数据的字节长度; block_name 为块名称; offset 为偏移量名称; 读取出的数据类型为 np.ndarray, dtype 可指定其中每个元素的类型。

4.2.3 测试 BRAM 读写

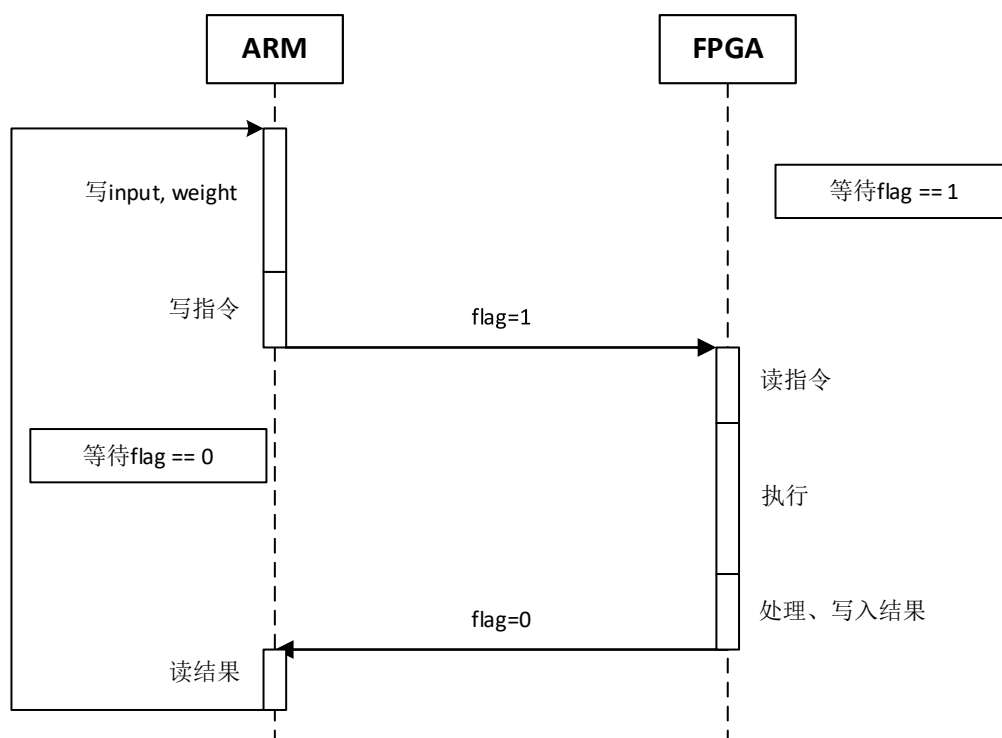
将 bram.py 拷贝至开发板/root 下并运行, 观察输出并理解样例中的写入方式和读取方式。

```
python3 bram.py
```

4.3 矩阵乘法软硬件交互设计

4.3.1 设计方案

ZYNQ 板子分为 ARM 侧(PS)和 FPGA 侧(PL), 交互设计如下图所示。



FPGA 侧负责实现矩阵乘法，ARM 侧负责传送输入数据 `input` 和权重数据 `weight`，接收返回的结果 `result`。下面介绍交互过程中的一些假设和具体流程。

假设：

- `input` 是一个输入矩阵，大小为 $(M \times N)$ ，类型为 `np.uint8`
- `weight` 是权重矩阵，大小为 $(N \times P)$ ，类型为 `np.int8`
- `result` 是结果矩阵，大小为 $(M \times P)$

指令(instr)格式：

[63:48]	[47:32]	[31:16]	[15:0]
null	input/weight N	weight P	input M

流程介绍：

1. ARM：将大小为 $(M \times N)$, (N, P) 的矩阵分别存入 `input` 和 `weight` 的 BRAM 块中。
2. ARM：按照指令格式构造指令，存入 `ir` 块对应的 `instr` 偏移位置中。
3. ARM：写入 `flag_01` 信号，即将 1 写入 `ir` 块地址对应的 `flag` 偏移位置中。（可参考 `bram.py` 下方样例）
4. FPGA：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：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：读结果，一次矩阵乘法结束。

4.3.2 pl_simulate.py

FPGA 侧操作实现在本实验中不涉及，使用 `pl_simulate.py` 模拟该过程，下面对该脚本进行介

绍：

由于硬件设计实现周期较长，修改也比较困难，因此在硬件实现以前，需要对整体设计进行验证，以确保设计的可行性。`pl_simulate.py` 作为一个简化的硬件模型，可以和 ARM 侧代码联合调试，用于验证硬件功能和软硬件交互逻辑的可行性，也可以作为 `golden_model` 用于 FPGA 侧代码功能的验证。相比于真实的硬件，`pl_simulate.py` 忽略了诸如寄存器、状态机、计算单元等硬件实现细节，只关注硬件的功能。

`pl_simulate.py` 通过单独的进程，以轮询的方式监测 `bram` 的指令和数据变化，并根据硬件的流程来读取数据、计算并写回计算结果，来完全模拟硬件的行为。其主体是一个 `While (true)` 循环，其具体工作流程如下：

1. 循环读取 `BRAM` 内的指令，等待新的指令写入。
2. 读取新指令，从中解析出要计算的权重和特征图矩阵大小。
3. 从 `BRAM` 中读取 `input` 和 `weight` 矩阵，执行矩阵乘法，并将结果写会 `output` 对应的 `BRAM` 块内。
4. 写入 `flag_00` 信号，以通知 ARM 端计算完成。

4.4 矩阵乘法软件接口 Matmul 实现

同学们需要在 `Matmul.py` 中实现矩阵乘法 `Matmul` 类，使得最后调用形式类似以下代码：

```
matmul = Matmul()

input = np.ones((m, n), dtype=np.uint8)

weight = np.ones((n, p), dtype=np.int8)

matmul(input, weight)
```

4.4.1 Matmul 类模板介绍

`Matmul` 类模板中预设了一些方法，同学们可以根据自身需求增加、修改、删除方法。下面进行详细的类方法介绍（更多参数说明可见代码注释）。

1) `__init__(self)`

类初始化变量，模板中仅设置了一个必须的参数，`self.systolic_size = 4`，该参数用于表示脉动阵列大小。`M`, `P` 的值需要满足该参数的倍数，不满足则将矩阵 `input/weight` 补零。

2) `__call__(self, input: np.uint8, weight: np.int8)`

实现直接调用实例的方法，输入 `input` 和 `weight`。

3) `send_data(self, data, block_name, offset='default')`

实现写入 `input` 和 `weight` 数据至 `BRAM` 的方法。输入 `data`，目标写入的块名称和偏移地址名称。

需要注意，`data` 需要根据 `self.systolic_size = 4` 进行补零。假设两个矩阵分别是 $(m,n) \times (n,p)$ ，`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：（补零原因可见 4.4.3 小节说明）

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

- 4) `send_instr(self, m, p, n)`
实现指令构建并写入指令至 BRAM，参数为矩阵大小。
- 5) `send_flag(self)`
实现 `flag=1` 信号写入
- 6) `recv_output(self, output_shape: tuple)`
实现结果接收，即监测 `flag` 信号并从 BRAM 中读取结果，需要将结果 `reshape` 为目标 `shape` 后返回。

4.4.2 实现注意事项

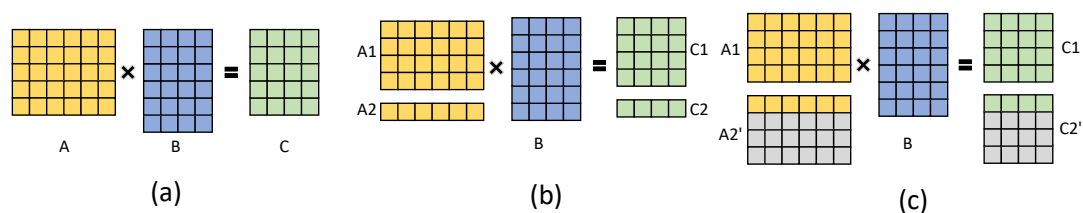
注意事项：

1. `input` 为 `np.uint8` 类型，`weight` 为 `np.int8` 类型，这是为了后续神经网络矩阵计算调用该接口
2. `input`、`weight` 需要进行补零

4.4.3 input 和 weight 补零原因

由于脉动阵列的尺寸 `self.systolic_size` 为 4，所以在脉动阵列本身只能支持结果矩阵大小为 `4x4` 的运算。为了支持更大规模的矩阵运算，硬件首先会对矩阵进行分块，然后分别在每个小块内部执行矩阵乘法。对于分块边缘处的情况，如果结果矩阵大小不能够满足 `4x4`，则必须将其进行补充一定的补充。

下图是一个矩阵补 0 的示例。如(a)所示，(5x6)的矩阵 A 与(6x4)的矩阵 B 进行相乘，得到(5x4)的矩阵 C。由于结果矩阵 C 已经超过了脉动阵列(4x4)的大小，所以需对 C 进行分块运算，如(b)所示。然而，分块后的 C2 矩阵小于脉动阵列所支持的大小，所以在其下方补充一些数据(灰色块)，使其大小变为 `4x4`，如图(c)所示。对应地，A2 矩阵也需要进行补充。



4.4.4 矩阵乘法测试

完成后运行 `Matmul.py` 和 `pl_simulate.py` 测试接口，测试样例写在该文件最下方。

(`pl_simulate.py` 介绍可见 4.3.2 小节)

1) 安装 `tmux`

为了同时在板子上开启两个终端分别运行两个测试文件，需要安装 `tmux` 终端复用器。

(`update` 过程中出现的 `warning` 和 `error` 可忽略)

```
apt update
apt install tmux
```

2) 开启一个新的 `tmux` 窗口并运行 `pl_simulate.py` (访问 BRAM 需要 `root` 权限，因此使用 `root` 用户运行)

```
tmux
```

```
python3 pl_simulate.py
```

3) 脱离 tmux 终端

在 tmux 的终端中先按下 **ctrl+b** 后松开，再按下 **d** 即可脱离。

（想要重新 attach 终端可使用 **tmux attach** 命令，具体使用方法可以百度）

4) 运行 Matmul.py 测试矩阵乘法

```
python3 Matmul.py
```

若运行正确可以看到以下结果：

```
~~~ demo1 pass ~~~
```

```
~~~ demo2 pass ~~~
```