Lab-5：设计 NaiveTPU 并行加速矩阵乘法

1 实验目标

1) 理解矩阵乘法的拆分，掌握控制流程的设计；
2) 理解模块设计思想，掌握自顶向下的设计方法；
3) 学习 FPGA 设计和实现流程。

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 Vivado 工程文件可在北航盘下载，其中包括两部分：
   a) 基础工程文件压缩包
      https://bhpan.buaa.edu.cn/link/AAB2C0D49E361A4CDDB9FE68BCC0173795
      Name: Lab-5：设计 NaiveTPU 并行加速矩阵乘法（基础工程文件）.rar
      Expires: 2025-03-01 16:04
      Pickup Code: mNz1
   b) 问答题工程文件压缩包
      https://bhpan.buaa.edu.cn/link/AA70D7ADD2A00D45BFBEBD11889C94AEFA
      Name: Lab-5：设计 NaiveTPU 并行加速矩阵乘法（问答题工程文件）.zip
      Expires: 2025-03-01 16:05
      Pickup Code: 3GcW
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 4.3 问题中的问题**。实验报告命名：学号+姓名+实验五。（实验报告撰写细节可参考“实验报告撰写格式”）
2) 将实验报告与工程文件压缩包，按时交至 SPOC 课程作业处。

4 实验内容

4.1 矩阵乘法模块的设计

4.1.1 设计目标

对于 FPGA 设计而言，最重要的，是确定设计目标，即所设计的模块需要完成什么样的功能，其次才是如何实现这个功能。根据实验需求，在 PS 侧完成卷积运算 img2col 的过程，在 PL 侧完成矩阵乘法运算，并将结果返回给 PS 侧。因此，PL 侧需要完成的目标即，矩阵的

乘法运算，且输入矩阵的大小可变。

按照设计，输入矩阵 FM 和 WM 分别满足以下特性：

矩阵 Feature（或称 Input），大小为 M×N，其元素均为 uint8 格式（无符号 8 位宽整型数据）；矩阵 Weight，大小为 N×P，其元素均为 sint8 格式（有符号 8 位宽整型数据）。输出矩阵 Output，大小为 M×P，其元素均为 sint32 格式（有符号 32 位宽整型数据）。

4.1.2 软硬件接口设计

一般而言，PS 侧和 PL 侧分别由软硬件人员各自设计。因此，软硬件接口需要在一开始首先约定好。为简化设计，采用 BRAM 完成 PS 侧和 PL 侧的交互，即，数据和控制指令均通过 BRAM 传输。在以下的说明中，PS 侧和 ARM 为同义词，PL 侧和 FPGA 为同义词，不再区分。

依据需求，共需要 4 个 BRAM，分别存储输入的 feature 矩阵，输入的 weight 矩阵，控制指令，输出的 output 结果矩阵。将他们分别命名为：BRAM_FM32b、BRAM_WM32b、BRAM_CTRL32b、BRAM_OUT32b。其中，BRAM_FM32b 用于存储输入的 feature 数据，BRAM_WM32b 用于存储输入的 weight 数据，BRAM_CTRL32b 用于存储控制信号，BRAM_OUT32b 用于存储矩阵乘法输出结果。这里，PS 侧和 PL 侧采用 AXI4LITE 协议进行通信，数据位宽为 32bit，使用的 BRAM 设置为 BRAM Controller 模式，相应为 32bit 位宽。

约定交互流程如下：

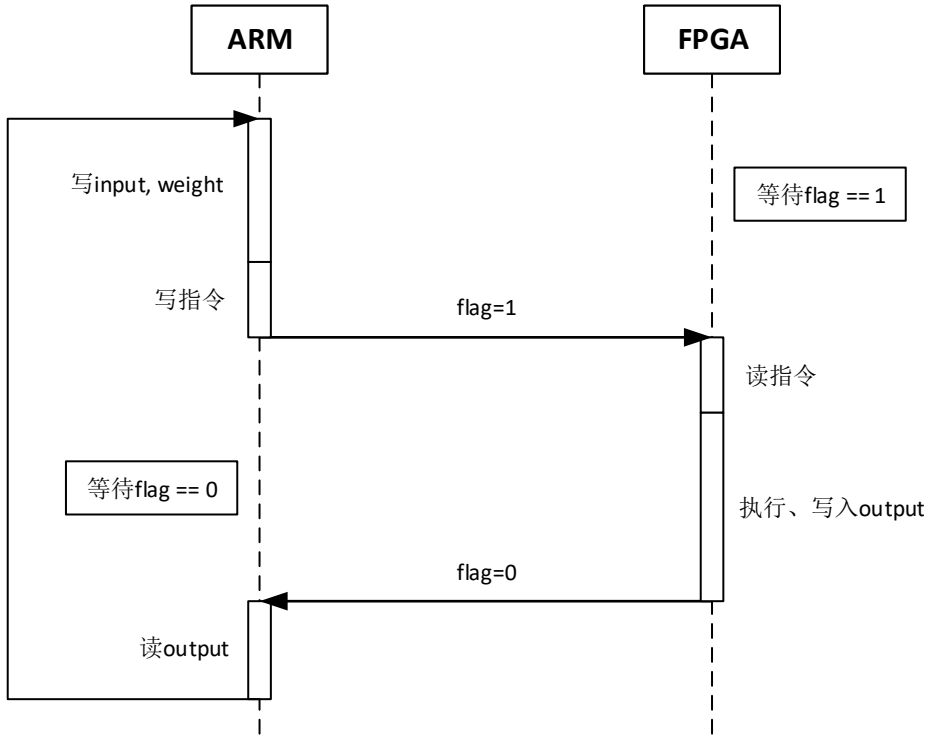

图 1 ARM 和 FPGA 交互流程图

如图 1 ARM 和 FPGA 交互流程图。ARM 首先将 input 数据（即 feature）、weight 分别存储在 BRAM_FM32b 和 BRAM_WM32b 中，之后将矩阵参数、指令 flag=1 分别写入 BRAM_CTRL32b 对应地址，等待 flag 为 0。FPGA（即 PL 侧）循环检测 flag 信号，当检测到 flag 为 1 时，读取矩阵参数，开始执行。FPGA 计算完毕将结果写入 BRAM_OUT 后，将 flag 置 0，进入等待 flag 为 1 的状态。ARM 检测到 flag 为 0 后，执行后续处理。

经过测试，MLP 和 LeNet 输入矩阵的大小最大如下：

表格 1：MLP 与 LeNet 网络模型输入矩阵最大规模

\	MLP 模型	LeNet 模型
Feature	(1, 784)	(784, 25)
Weight	(784, 100)	(400, 120)

依据表格 1：MLP 与 LeNet 网络模型输入矩阵最大规模，我们约定 BRAM 地址映射如下：

表格 2：BRAM 地址映射

起始地址	大小（字节）	读/写	描述
0x4000_0000	32K	r/w	Feature mem
0x4002_0000	128K	r/w	Weight mem
0x4004_0000	32K	r/w	Output mem
0x4006_0000	4K	r/w	FLAG：0x0-0x04 (arm 读写，PL 读写) 指令：0x10-0x17 (arm 写，PL 读)

输入矩阵的 BRAM 大小的计算方式为：

Feature 矩阵最大为(784, 25)，一个数据为 8bit，因此需要 BRAM 大小为 784×25 字节，取 32KB。Weight 矩阵最大为(784, 100)，一个数据为 8bit，因此需要 BRAM 大小为 784×100 字节，取 128KB。

表格 2：BRAM 地址映射中，指令含义如下：

表格 3 指令 M*N（input）×N*P（weight）

范围	描述
[15:0]	Feature M
[31:16]	Weight P
[47:32]	Feature/Weight N
[63:48]	NULL

4.1.3 矩阵乘法模块设计

确定模块功能后，需要考虑如何实现本模块的功能。由于输入矩阵的大小不确定，本模块采用矩阵分块的方式，将输入矩阵拆分成小矩阵来分别进行计算。

设A为$m \times l$矩阵，B为$l \times n$矩阵，分块成

$$A = \begin{bmatrix} A_{11} & \dots & A_{1t} \\ \dots & & \dots \\ A_{s1} & \dots & A_{st} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & \dots & B_{1r} \\ \dots & & \dots \\ B_{t1} & \dots & B_{tr} \end{bmatrix},$$

其中A_{i1}，A_{i2}，…，A_{it}的列数分别等于B_{1j}，B_{2j}，…，B_{tj}的行数，那么

$$AB = \begin{bmatrix} C_{11} & \dots & C_{1r} \\ \dots & & \dots \\ C_{s1} & \dots & C_{sr} \end{bmatrix},$$

其中

$$C_{ij} = \sum_{k=1}^{t} A_{ik} B_{kj} \, (i = 1, \dots, s; j = 1, \dots, r).$$

为了降低设计难度，我们假设输入矩阵分块时，均分块为同样大小的子矩阵。为了尽可能提高运算效率，子矩阵分块时，分块的大小应尽量大，这样所需要运算的块数变少，可以缩短整体的运算时间。但受限于 FPGA 资源，如果分块过大，有可能 FPGA 无法满足一次矩阵运算所消耗的资源数量。因此，需要对器件资源进行评估。

由于本次实验采用 ZYNQ 7020 开发板，其 LUT 资源为 53200，BRAM 资源为 140。结合

lab3 中的 MAC 模块，通过预综合方式（直接对某部分代码进行综合），得知一个 MAC 模块大概消耗 LUT 资源为 230，一个 Multiply_8x8 模块大概消耗 LUT 资源为 14720（230×64）。其中，Multiply_8x8 模块就是一个 8×8 大小的矩阵块的运算单元，其中包含 64 个 MAC 运算模块。

通过对资源的分析，我们得知，在这块板子上最多可以放下 3 个 Multiply_8x8 模块。但出于便于设计的考虑，同时也是为了预留一些资源给其他控制逻辑，确定使用两个 Multiply_8x8 模块同时进行矩阵乘法运算。此时，可以确定矩阵分块的大小为 8×16。

前面提到，PS 侧和 PL 侧采用 AXI4LITE 协议进行通信，数据位宽为 32bit，这意味着一个时钟只能传输 4 个数据（每个数据为 8bit）。而对于 8×16 的子矩阵运算单元（两个 Multiply_8x8 模块），每个时钟两个输入接口分别需要输入 8 个数和 16 个数。当我们将 PS 侧传来的输入矩阵存入 32bit 位宽的 BRAM 后，如果不进行位宽转换，那么该 BRAM 在一个时钟下也只能输出 32bit，无法满足 8×16 的子矩阵运算单元的需求，这会降低运算效率。因此，当输入矩阵存入 32 位宽的 BRAM 后，还应分别对其进行位宽转换，放入 64 位宽和 128 位宽的 BRAM 当中。

对于每个子矩阵的运算，其本质过程就是将输入搬移到 8×16 的子矩阵运算单元，再将输出结果搬走。由于矩阵分块有可能引入一些“残余的块”，它们会被填充成相同的大小，因此在搬移输出结果的时候，需要知道有效数据部分的大小。此外，在每次对子矩阵进行运算的时候，都需要知道计算的子矩阵位于原矩阵的哪个部分。还需要知道什么时候可以判定原矩阵的乘法已经计算完毕，即所有子矩阵都已经完成运算。

采用自顶向上的设计思想。首先需要有一个整体的控制模块，在开始运算前，将输入矩阵进行位宽转换。开始运算时，对输入矩阵进行分块，控制参与本次子矩阵运算的是哪两个“块”，并控制将输出结果裁剪为合适的大小，搬移到合适的位置。之后，需要有具体完成位宽转换功能的模块，负责将输入的矩阵格式转换为需要的矩阵格式；需要有具体完成子矩阵相乘的模块，负责将子矩阵数据搬移到运算模块，并控制本次子矩阵运算结果的裁剪；需要有具体完成子矩阵结果拼接的模块，负责依据本次运算子矩阵所处的位置，将结果放在输出 BRAM 的正确地址。而 8×16 的子矩阵运算单元，是由两个 Multiply_8x8 模块拼接而成的。出于上述考虑，设计矩阵乘法模块结构如图 2 矩阵乘法模块框图所示。

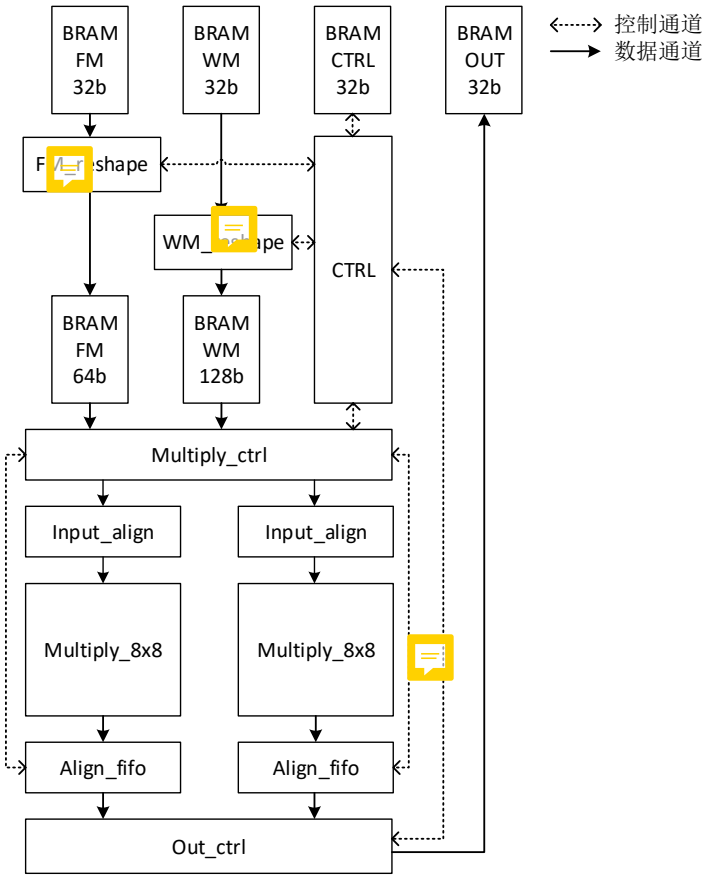

图 2 矩阵乘法模块框图

如图 2 矩阵乘法模块框图，CTRL 模块是 PL 侧的总控制模块。初始状态下，等待 flag 信号为 1，该信号从 BRAM_CTRL 中读取。检测到 flag 为 1 后，分别通知 FM_reshape 和 WM_reshape 开始工作，并读取计算参数。

FM_reshape 负责将 feature 数据重构，存储到 BRAM_FM64b 中，其数据位宽为 64bit。WM_reshape 负责将 weight 数据重构，存储到 BRAM_WM128b 中，其数据位宽为 128bit。将数据进行重构而不直接读取原本 BRAM 中数据进行计算的原因是，BRAM 在一个时钟周期内只能访问一个地址的数据，如果读取 32bit 位宽的 BRAM 作为 Multiply_8x8 的输入，需要连续读取两个时钟以拼接为 64bit（一个数据 8bit，8 个数据 64bit），这会影响 Multiply_8x8 模块的计算。在对原矩阵进行计算时，是通过将原矩阵拆解为子矩阵进行计算的，feature 和 weight 中的子矩阵会被多次访问。因此，在首次得到数据后，将其进行拼接，主要目的是避免后续多次访问时，每次都要进行拼接操作。

在 FM_reshape 和 WM_reshape 完成工作后，通知 CTRL 模块。CTRL 模块首先依据计算参数，对原矩阵进行拆解，将子矩阵的数据存储地址送给 Multiply_ctrl 模块，通知其开始执行矩阵计算。Multiply_ctrl 模块依据子矩阵数据地址，从 BRAM_FM_64b 和 BRAM_WM_128b 中搬移数据，分别送给两个 Multiply_8x8 模块进行子矩阵运算。其中，BRAM_FM_64b 的数据同时送给两个 Multiply_8x8 模块，BRAM_WM_128b 的数据高低 64bit 分别送给两个 Multiply_8x8 模块。

Multiply_8x8 模块中有 8×8 个 MAC 模块，每个 MAC 模块负责进行向量的乘累加运算（即向量乘）。因此，各 MAC 模块得出的即为输出矩阵的一个元素。这些输出元素存储在 Align_fifo 中，Align_fifo 实际上包括 8 个 fifo，每个 fifo 存储输出矩阵的一列元素。

Multiply_ctrl 模块将子矩阵相乘运算完成后，反馈给 CTRL 模块。CTRL 模块将该子矩阵结果在输出矩阵中的存储地址送给 Out_ctrl 模块。Out_ctrl 模块将输出结果搬移到 BRAM_OUT

中。搬移完毕后，通知 CTRL 模块，由 CTRL 模块启动下一个子矩阵的运算。

待 CTRL 模块判定所有子矩阵计算完毕后，置 flag 信号为 0，等待 ARM 启动下一次的矩阵运算。

4.1.4 数据格式约束

依据图 2 矩阵乘法模块框图，需要对矩阵数据在各个 BRAM 中的存储形式进行约束。**“矩阵乘法模块中的数据形式.xlsx”**中两个矩阵 A、B 为例说明数据在各个 BRAM 中的存储形式。设定矩阵大小如下：

A：M×N

B：N×P

输出：M×P

举例：M——10，N——30，P——20

另外，**“NaiveTPU_硬件矩阵乘法原理.pptx”**中的**数据形式（举例）**部分也可以帮助理解。

下面对子矩阵地址计算进行部分说明。

定义符号“//”：a//b 表示 a 除 b 的商向下取整。

对于 M×N 的矩阵，在 BRAM_FM_64b 中的存储，共有[(M-1)//8+1]个子矩阵，相邻子矩阵首地址偏移量为 1，对于单个子矩阵，在 BRAM_FM_64b 中，相邻数据地址偏移为[(M-1)//8+1]（子矩阵个数和单个子矩阵内相邻数据地址偏移一致）。

对于 N×P 的矩阵，在 BRAM_WM_128 中的存储，共有[(P-1)//16+1]个子矩阵，相邻子矩阵首地址偏移量为 1，对于单个子矩阵，在 BRAM_WM_128b 中，相邻数据地址偏移为[(P-1)//16+1]（子矩阵个数和单个子矩阵内相邻数据地址偏移一致）。

在计算两个矩阵相乘时，首先将其拆成子矩阵进行计算。规定子矩阵的计算顺序为先横向，再纵向。例如，FM（M×N）有子矩阵 FM1 和 FM2，WM（N×P）有子矩阵 WM1 和 WM2，那么，计算子矩阵的顺序为：FM1×WM1，FM1×WM2，FM2×WM1，FM2×WM2。

对于输出矩阵，已知其大小为 M×P，每个子矩阵均不大于 8×16 的规模。对于每个子矩阵中的行，其相邻数据偏移为 1，对于每个子矩阵中的列，其相邻数据偏移为 P。对于行方向的子矩阵，其对应位置数据地址偏移相差 16；对于列方向的子矩阵，其对应位置数据地址偏移相差 8*P。

例如，“矩阵乘法模块中的数据形式.xlsx”中，输出矩阵子矩阵 2 数据 0c16 和 0c17 在同一行且相邻，在 BRAM_OUT_32b 中的地址差 1；子矩阵 2 数据 0c16 和 1c16 在同一列且相邻，地址相差 P=20；子矩阵 1 和子矩阵 2 为行方向相邻子矩阵，对应位置数据，如 0c0 和 0c16，地址相差 16；子矩阵 1 和子矩阵 3 为列方向相邻子矩阵，对应位置数据，如 0c0 和 8c0，地址相差 8*P =160。

4.1.5 矩阵乘法模块各子模块设计

按照图 2 矩阵乘法模块框图，还需要将各个模块功能进行细化。为降低设计复杂度，主要采用状态机进行控制模块的设计。

4.1.5.1 总控制模块 CTRL

根据前面描述的 FPGA 数据处理流程，CTRL 模块状态机分为以下状态：

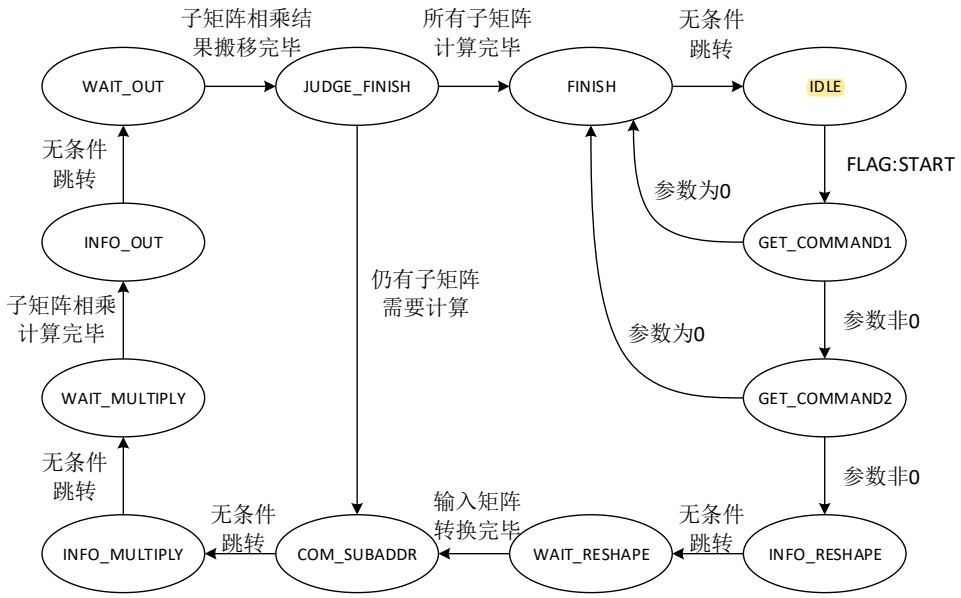

图 3 CTRL 模块状态转移图

IDLE：初始状态，循环读取控制 BRAM_CTRL 中的 FLAG 信号，直到该信号为 1，跳转到 GET_COMMAND 状态。

GET_COMMAND：读取 BRAM_CTRL 中的指令，获取指令参数 M、N、P，跳转到 INFO_RESHAPE 状态。如果发现指令异常（M、N、P 任一为 0），则跳转到 FINISH 状态。该状态拆分成 GET_COMMAND1（获取 M、P）和 GET_COMMAND2（获取 N）来执行。

INFO_RESHAPE：通知 FM_reshape 和 WM_reshape 依据指令参数 M、N、P 开始工作，跳转到 WAIT_RESHAPE 状态。

WAIT_RESHAPE：等待 FM_reshape 和 WM_reshape 完成。之后跳转到 COM_SUBADDR 状态。

COM_SUBADDR：依据指令参数 M、N、P 以及当前子矩阵位置计算下一个子矩阵在 BRAM_FM 和 BRAM_WM 中的存储地址，并计算该子矩阵在 BRAM_OUT 中的存储地址。跳转到 INFO_MULTIPLY 状态。

INFO_MULTIPLY：通知 Multiply_ctrl 模块按照子矩阵在 BRAM_FM 和 BRAM_WM 中的存储地址开始工作，跳转到 WAIT_MULTIPLY 状态。

WAIT_MULTIPLY：等待 Multiply_ctrl 模块完成工作。之后跳转到 INFO_OUT 状态。

INFO_OUT：通知 Out_ctrl 模块按照子矩阵在 BRAM_OUT 中的存储地址开始工作，跳转到 WAIT_OUT。

WAIT_OUT：等待 Out_ctrl 模块完成工作。之后跳转到 JUDGE_ FINISH 状态。

JUDGE_FINISH：判断当前子矩阵是否为最后一个子矩阵。若是，则跳转到 FINISH 状态；若不是，则跳转到 COM_SUBADDR 状态。

FINISH：修改 FLAG 标记，跳转到 IDLE 状态。

4.1.5.2 FM 数据重组模块 FM_reshape

IDLE：初始状态，等待 CTRL 模块通知开始工作。之后，跳转到 COM 状态。

COM：依据输入的 M、N 计算需要执行的次数。其中，M 是 BRAM_FM_32b 中每次拼接前数据的个数，M（＞0）个数需要在 BRAM_FM_32b 中存储[(M-1)//4+1]行。在 BRAM_FM_64b 中，则需要存储[(M-1)//8+1]行。之后，跳转到 WORK 状态。

WORK：执行搬移操作。以[(M-1)//4+1]为第一级循环，N 为第二级循环，将 BRAM_FM_32b 中的数据通过拼接搬移到 BRAM_FM_64b 中。在拼接时，若不足一行的拼接，需要补 0。如

果遇到 FM 矩阵新行，此时无需进行拼接，补 0 即可。完成搬移后，跳转到 FINISH 状态。

FINISH：反馈给 CTRL 模块，告知已完成搬移。之后，跳转到 IDLE 状态。

4.1.5.2 WM 数据重组模块 WM_reshape

IDLE：初始状态，等待 CTRL 模块通知开始工作。之后，跳转到 COM 状态。

COM：依据输入的 P、N 计算需要执行的次数。其中，P 是 BRAM_WM_32b 中每次拼接前数据的个数，P（＞0）个数需要在 BRAM_WM_32b 中存储[(P-1)//4+1]行。在 BRAM_WM_128b 中，则需要存储[(P-1)//16+1]行。之后，跳转到 WORK 状态。

WORK：执行搬移操作。以[(P-1)//4+1]为第一级循环，N 为第二级循环，将 BRAM_WM_32b 中的数据通过拼接搬移到 BRAM_WM_128b 中。在拼接时，若不足一行的拼接，需要补 0。如果遇到 WM 矩阵新行，此时无需进行拼接，补 0 即可。完成搬移后，跳转到 FINISH 状态。

FINISH：反馈给 CTRL 模块，告知已完成搬移。之后，跳转到 IDLE 状态。

4.1.5.3 子矩阵相乘控制模块 Multiply_ctrl

IDLE：初始状态，等待 CTRL 模块通知开始工作。之后，跳转到 INFO 状态。

INFO：告知 Align_fifo 本次运算子矩阵的规模（从 CTRL 模块获取），以免 Align_fifo 将多余的 0 写入。之后，跳转到 WORK 状态。

WORK：依据从 CTRL 模块获取的数据个数 N、子矩阵首数据地址、子矩阵地址递增量（[(M-1)//8+1]和[(P-1)//16+1]），执行子矩阵相乘的逻辑控制。其本质上，是将数据搬移送入 Multiply_8x8 模块。完成搬移后，跳转到 WAIT 状态。

WAIT：等待 Align_fifo 模块获取子矩阵相乘结果。当 Align_fifo 模块获取结果后，产生 align_fifo_get_all 信号通知本模块。两个 Align_fifo 模块均获取运算结果后，跳转到 FINISH 状态。

FINISH：反馈给 CTRL 模块，告知已完成子矩阵相乘运算。之后，跳转到 IDLE 状态。

4.1.5.4 子矩阵输出缓存模块 Align_fifo

依据 Multiply_ctrl 模块告知的子矩阵运算结果规模，将输出数据分别存储在 8 个 FIFO 当中。在接收数据时，需要对数据进行判断是否为所需要的数据，因为 Multiply_8x8 模块输出的总是 8×8 规模的子矩阵，其中包含了一些不需要的 0 数据。由于需要的子矩阵结果总是存在于 8×8 子矩阵的左上角，不需要的填充 0 数据均在右侧和下侧，因此依据本次子矩阵结果规模，以及一些计数器，就可以判定哪些数据是实际需要的。

当收集到足够数据后，将 align_fifo_get_all 信号置 1，告知 Multiply_ctrl 模块。之后等待 Out_ctrl 模块控制输出（通过 out_ctrl_ready 信号）。输出时，依次轮询各个 FIFO，但不能超过本次子矩阵的大小。

4.1.5.5 输出控制模块 Out_ctrl

IDLE：初始状态，等待 CTRL 模块通知开始工作。之后，跳转到 WORK 状态。

WORK：依据从 CTRL 模块获取的子矩阵首数据坐标（Ma、Pa），结果矩阵规模（P），子矩阵结果大小（sub_P，sub_M），将输出子矩阵结果搬移到 BRAM_OUT_32b 中。之后，跳转到 FINISH 状态。

FINISH：反馈给 CTRL 模块，告知已完成结果搬移。之后，跳转到 IDLE 状态。

4.1.5.6 Multiply_8x8 模块结构

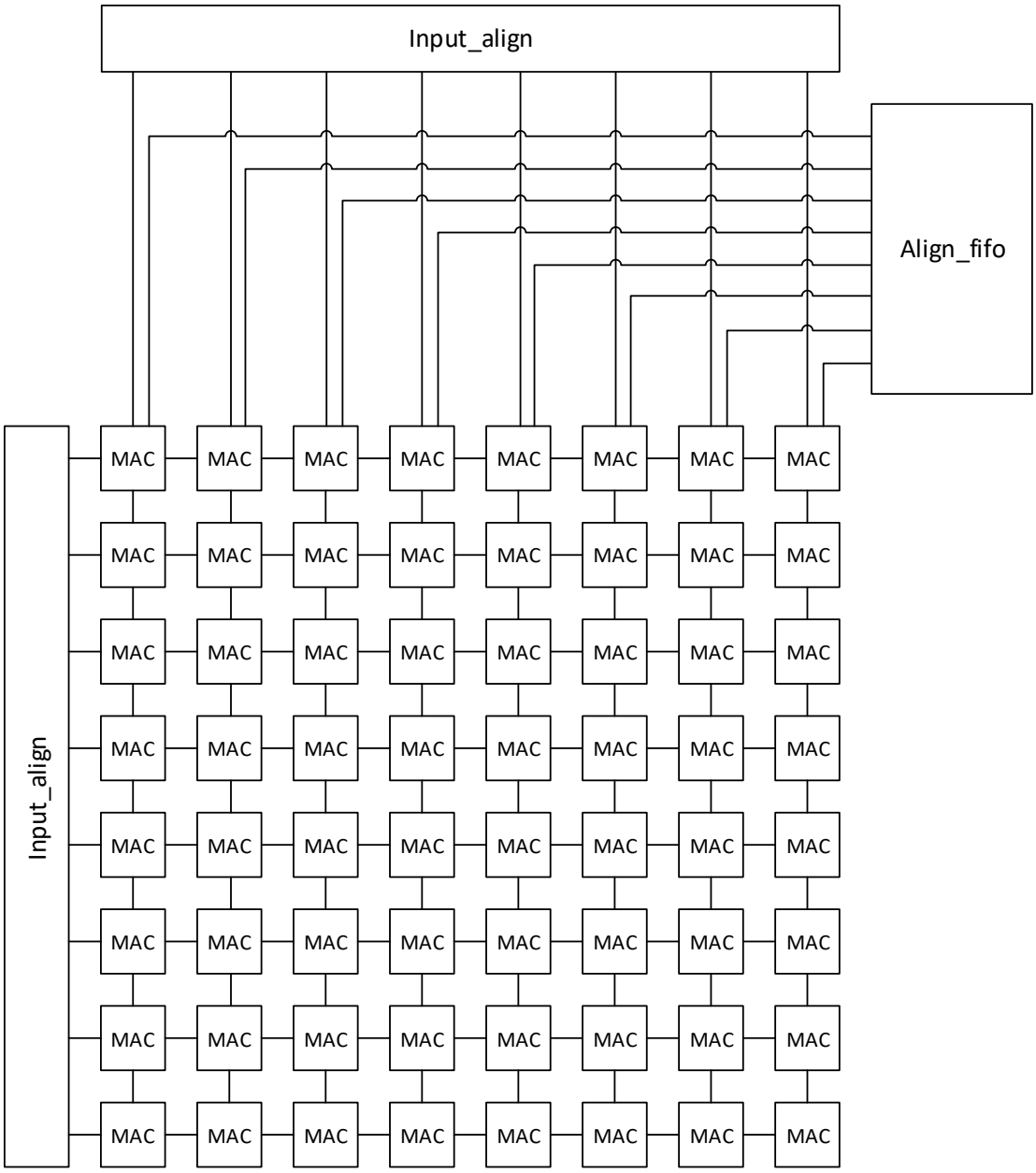

图 4 Multiply_8x8 模块框图

4.1.5.7 MAC 模块结构

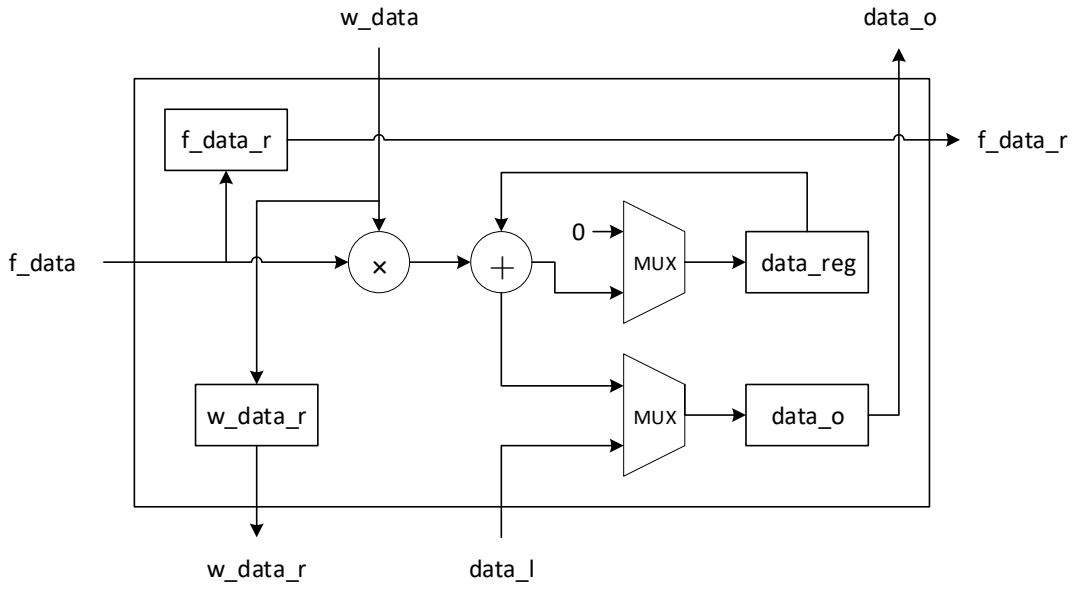

图 5 MAC 模块框图

4.2 矩阵乘法模块的实现

4.2.1 IP 核的使用

IP 核，即知识产权核，是“用于 ASIC 或 FPGA 中的预先设计好的电路功能模块”。调用 IP 核能避免重复劳动，大大减轻工程师的负担。在矩阵乘法模块的设计中，主要用到 FIFO 和 BRAM 的 IP 核。

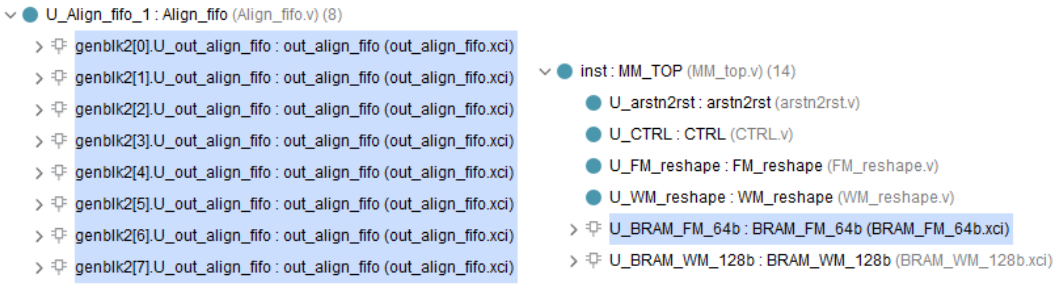

图 6 矩阵乘法模块中使用到的 IP 核（左图：FIFO，右图：BRAM）

关于它们的使用，网上很容易找到介绍，此处不再赘述。

4.2.2 仿真

4.2.2.1 仿真器的选择

在 Lab 3 中，使用了 Vivado 自带的仿真器。在菜单栏->Tools->Settings->Simulation->Target simulator 中可以选择其他仿真器。

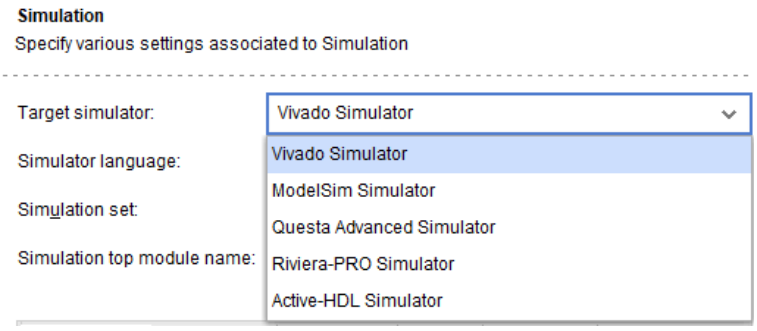

图 7 Vivado 中仿真器的选择

使用其他仿真器，同时需要在菜单栏->Tools->Settings->3rd Party Simulators 中选择安装目录以及对应的编译库路径。

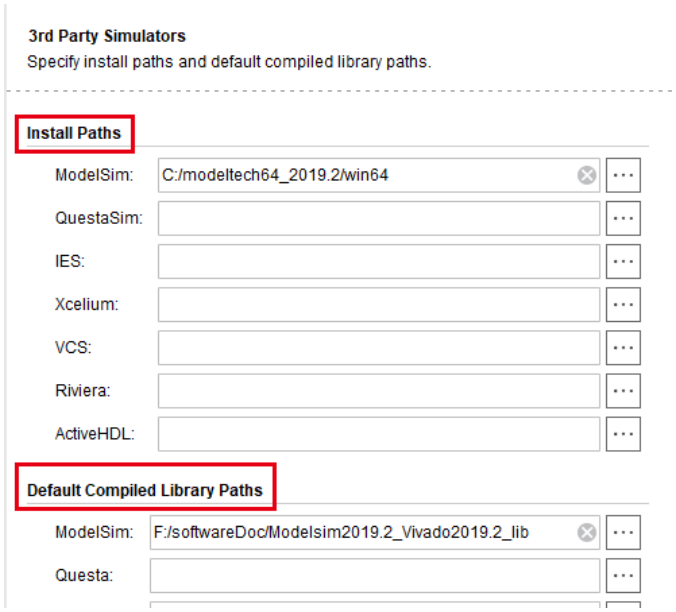

图 8 Vivado 中第三方仿真器设置

第三方仿真器不是必须的。如对 modelsim 仿真器有兴趣，可以搜索“vivado modelsim 联合仿真”词条。

但是**需要特别注意的是，不同的仿真器对同样的代码可能有不同的解释，尤其是阻塞赋值和非阻塞赋值**。因此，在写激励文件的时候，需要特别注意 initial 块中的“=”和“<=”，它们在不同的仿真器中可能存在不同的解释，往往表现为仿真出来的信号存在一拍的偏差。

例如，在 initial 块中使用“=”时，注意图中的 FM_reg_valid 信号：

```
206 initial begin
207     while(1) begin
208         file_FM = $fopen("E:/workspace/beihang_PE2/tb/test1/FM.txt","r");
209         FM_reg_valid = 1'b0;
210         FM_reg0 = 'b0;
211         FM_reg1 = 'b0;
212         FM_reg2 = 'b0;
213         FM_reg3 = 'b0;
214         wait(c_state==WRITE_FM);
215         while(!$feof(file_FM)) begin
216             @(posedge arm_clk)
217             FM_reg_valid = 1'b1;
218             line_FM = $fscanf(file_FM,"%d,%d,%d,%d,",FM_reg0,FM_reg1,FM_reg2,FM_reg3);
219         end
220         FM_reg_valid = 1'b0;
221         FM_reg0 = 'b0;
222         FM_reg1 = 'b0;
223         FM_reg2 = 'b0;
224         FM_reg3 = 'b0;
225         file_FM = $fopen("E:/workspace/beihang_PE2/tb/test2/FM.txt","r");
226         wait(c_state==WRITE_FM);
227         while(!$feof(file_FM)) begin
228             @(posedge arm_clk)
229             FM_reg_valid = 1'b1;
230             line_FM = $fscanf(file_FM,"%d,%d,%d,%d,",FM_reg0,FM_reg1,FM_reg2,FM_reg3);
231         end
232     end
233 end
```

图 9 FM_reg_valid 在 initial 中使用“=”

```
310 always @(posedge arm_clk or posedge rst) begin
311     if (rst) begin
312         arm_BRAM_FM32_wea <= 'b0;
313     end
314     else begin
315         arm_BRAM_FM32_wea <= {4{FM_reg_valid}};
316     end
317 end
```

图 10 将 FM_reg_valid 打拍

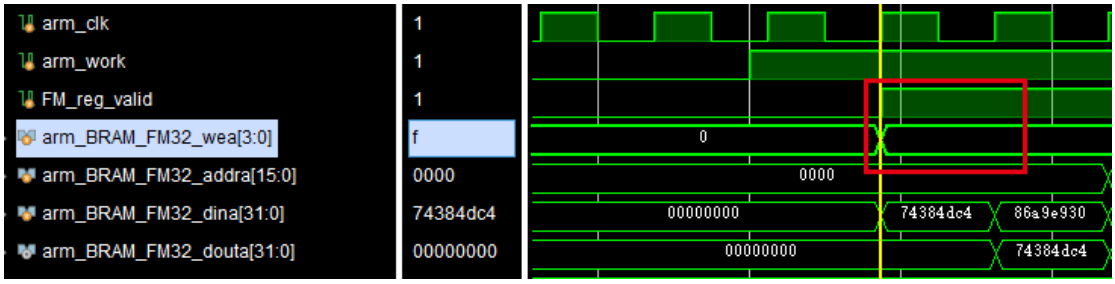

图 11 Vivado 仿真器结果

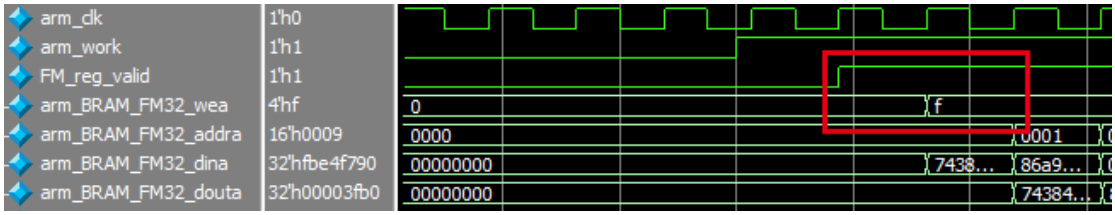

图 12 modelsim 仿真结果

可以看到，两个仿真器对阻塞赋值“=”的解释并不一致，尽管 FM_reg_valid 信号波形看起来是一致的，但对下一级的寄存器 arm_BRAM_FM32_wea 确实产生了影响。

鉴于两种仿真器的不同行为，在代码中做了一些区分，使用参数 SIMULATOR 进行选择。

```
310   parameter SIMULATOR = "VivadoSimulator"; // "VivadoSimulator" | "ModelsimSimulator"
311   // parameter SIMULATOR = "ModelsimSimulator";
```

图 13 参数 SIMULATOR

4.2.2.2 工程结构

本次实验所需要的工程示例已提供，可以直接下载使用（指的是基础工程文件压缩包）。本小节下述部分为操作提示，本工程已经完成，不需要再次执行。

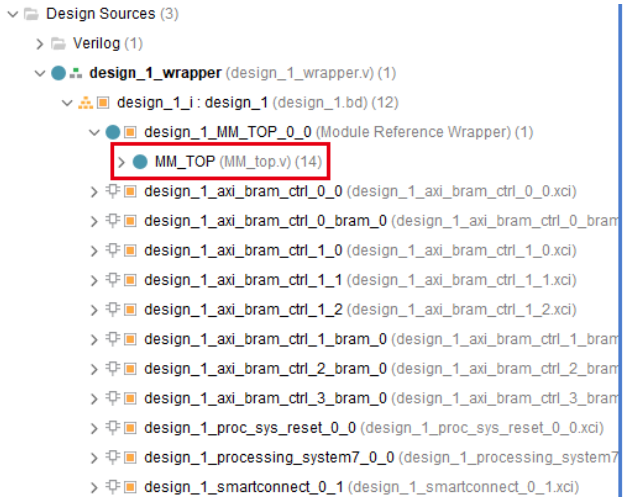

图 14 工程结构

如图 14 工程结构，MM_TOP 模块是矩阵乘法模块，其它模块均是在 Block Design 中添加的。在 Block Design 中，右键 MM_TOP 模块，选择 Add Module to Block Design 就可将自己设计的模块添加到其中。

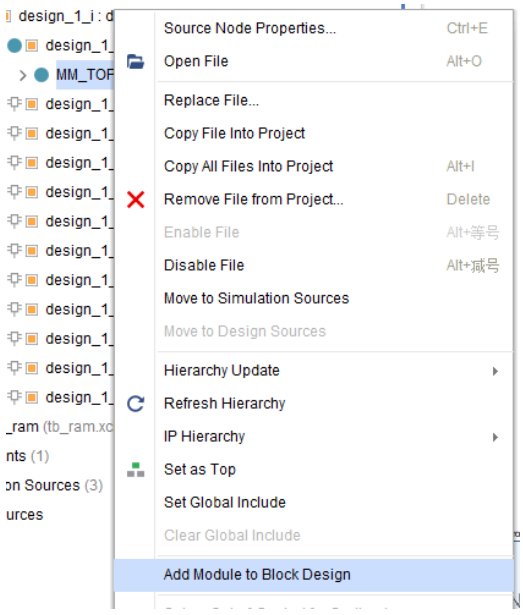

图 15 选择 Add Module to Block Design（本工程中已经添加）

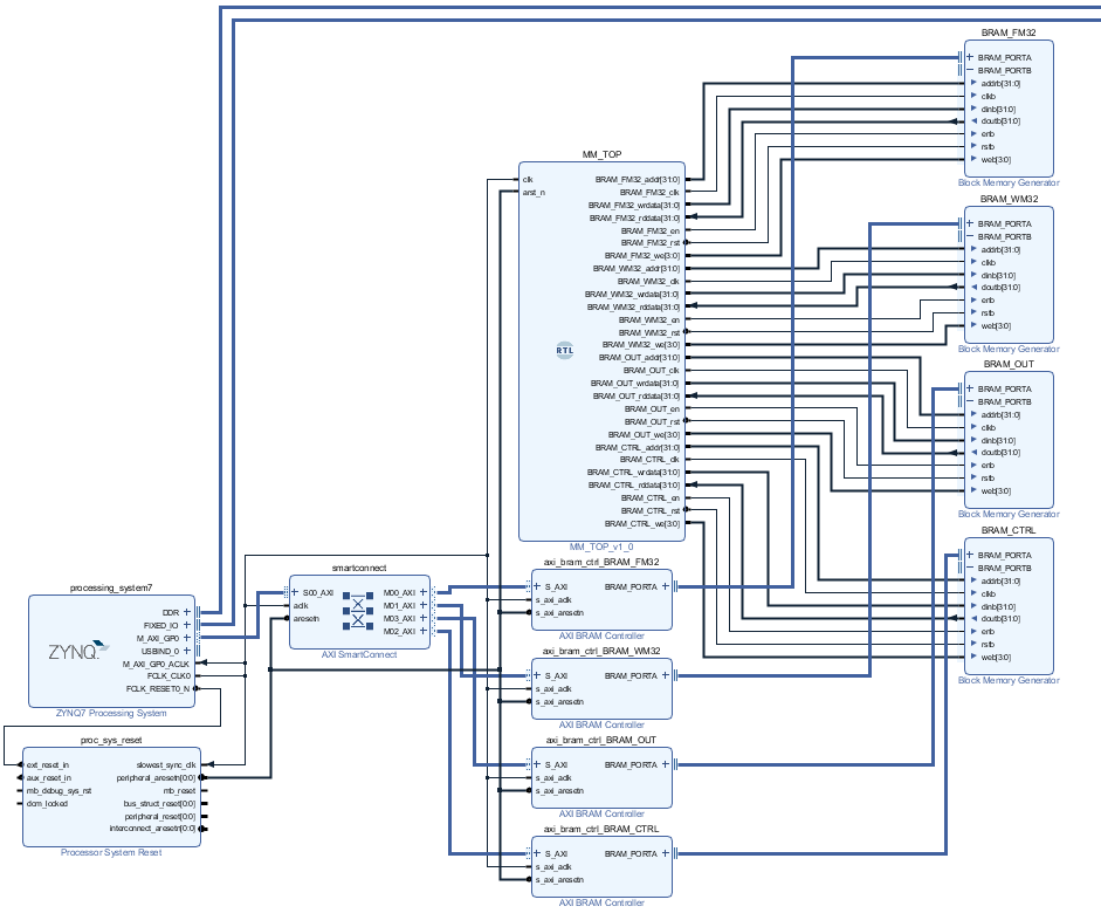

图 16 Block Design 连线图

顶层的 design_1_wrapper 文件可以由 Vivado 自动生成。其中，design_1 文件即创建的 Block Design 文件。

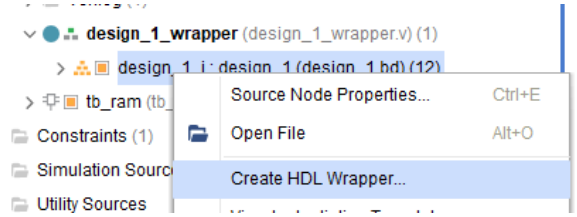

图 17 Vivado 自动生成顶层文件（本工程中已经添加）

4.2.2.3 仿真环境

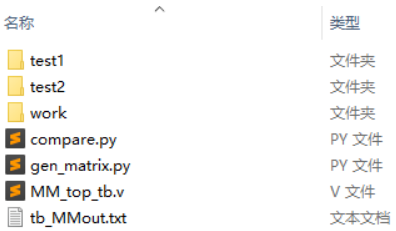

图 18 工程目录下 tb 文件夹

如图 18 工程目录下 tb 文件夹，test1 和 test2 文件夹是输入矩阵，由 gen_matrix.py 生成。Work 文件夹是文本编辑器产生的临时文件，不需要关注，可以删除。Compare.py 用于对比输入文件和仿真结果是否一致。Gen_matrix.py 用于产生随机输入矩阵。MM_top_tb.v 是

本模块的激励文件，该文件模拟了 ARM 的行为，将输入送入矩阵乘法模块，并将返回的结果写入 tb_MMout.txt 文件中。

打开 gen_matrix.py 文件，根据需要修改参数如下：

```
15   M1 = 10
16   N1 = 25
17   P1 = 6
18
19   M2 = 17
20   N2 = 33
21   P2 = 25
```

图 19 gen_matrix.py 需要设定参数

其中，M1、N1、P1 是输入 1 的参数，生成的文件保存在 test1 下；M2、N2、P2 是输入 2 的参数，生成的文件保存在 test2 下。对于每个输入，依据 M、N、P 产生元素值随机的 FM 矩阵和 WM 矩阵，并计算他们的矩阵乘结果，保存下来。

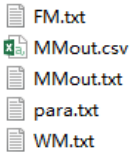

图 20 gen_matrix.py 生成文件

打开 MM_top_tb.v，**修改所有绝对路径为自己工程目录**。

```
1   `timescale 1ns/1ps
2   `include "E:/workspace/beihang_PE2/rtl/define.v"
```

图 21 类似路径有多处，注意全部修改

依据自己的仿真器，修改参数。

```
310   parameter SIMULATOR = "VivadoSimulator"; // "VivadoSimulator" | "ModelsimSimulator"
311   // parameter SIMULATOR = "ModelsimSimulator";
```

图 22 修改仿真器参数

之后，在 Vivado 中执行仿真。仿真激励会循环将 test1 和 test2 送入矩阵乘法模块，并将矩阵乘法模块的输出写入 tb_MMout.txt。

停止仿真后，执行 compare.py。如出现数据结果不匹配的情况，该脚本会将其打印。如提示“data right”，说明数据结果匹配，仿真正确。

4.2.3 调试和上板

4.2.3.1 抓信号

Vivado 提供抓取信号波形的功能，可以实现在线调试。该功能主要通过 ila 的 IP 核实现。下面介绍一种方法。

CTRL.v

```
1   `include "define.v"
2
3   module CTRL(
4       input clk,
5       (*mark_debug = "true"*) input rst,
6       // connect BRAM_CTRL
7       (*mark_debug = "true"*) output reg [31:0] BRAM_CTRL_addr,
8       output BRAM_CTRL_clk, // clk
9       (*mark_debug = "true"*) output reg [31:0] BRAM_CTRL_wrdata,
10      (*mark_debug = "true"*) input [31:0] BRAM_CTRL_rddata,
11      (*mark_debug = "true"*) output BRAM_CTRL_en, // 1'b1
12      (*mark_debug = "true"*) output BRAM_CTRL_rst, // rst
13      (*mark_debug = "true"*) output reg [3:0] BRAM_CTRL_we,
```

图 23 在需要抓取的信号前添加(*mark_debug = "true"*)，保存文件

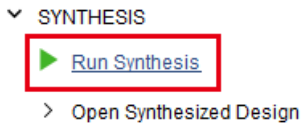

图 24 点击综合，等待综合完成

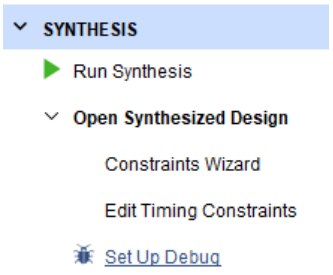

图 25 点击 Set Up Debug

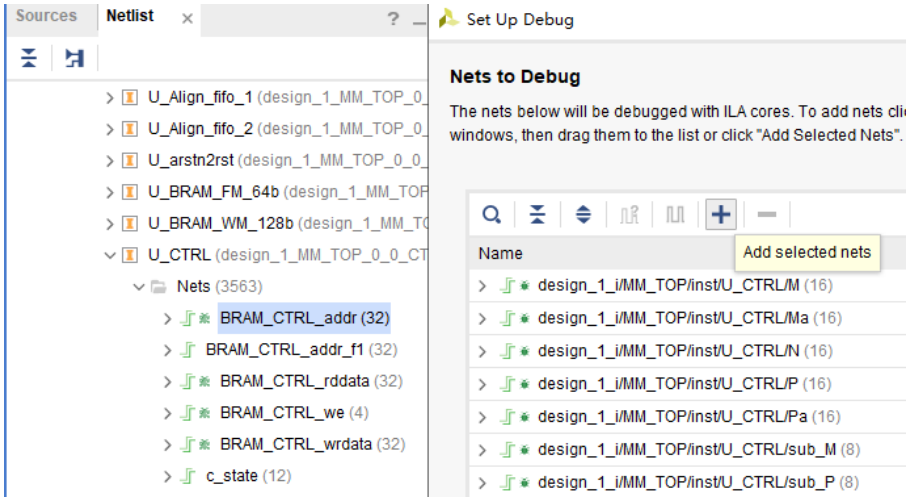

图 26 在 Netlist 中找到相应信号，点击 Add selected nets 进行添加

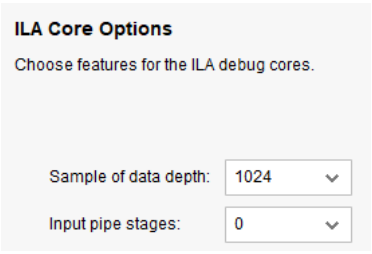

图 27 设置采样深度和流水线级数，可不修改

点击 Ctrl+S 保存综合修改。

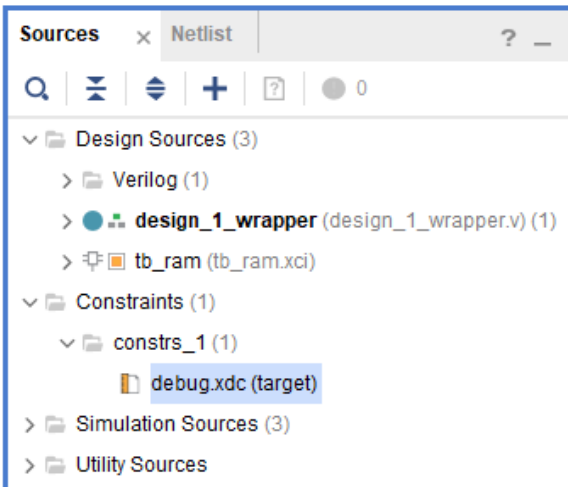

图 28 打开 debug.xdc 可以看到添加的 ila 核

4.2.3.2 上板

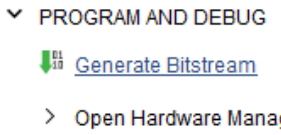

图 29 直接生成 bit 文件（会提示首先进行 Implementation）

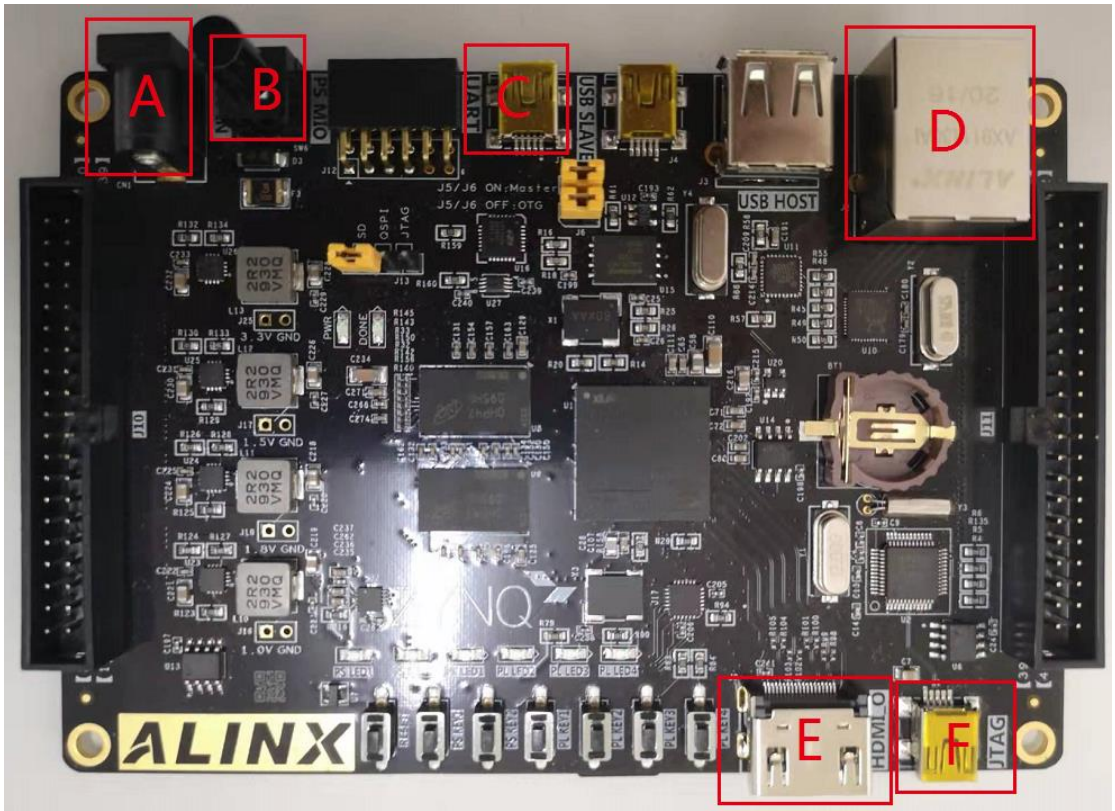

图 30 A（电源口）；B（板子开关）；C（UART 口，用于串口通信）；D（网口，用于网口通信）；E（HDMI 口，用于外接显示器）；F（JTAG 口，用于下载 bit 文件和在线调试）

板子和 PC 进行连线。

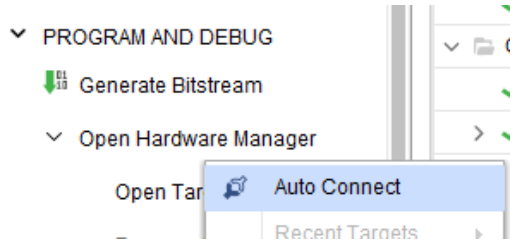

图 31 连接器件

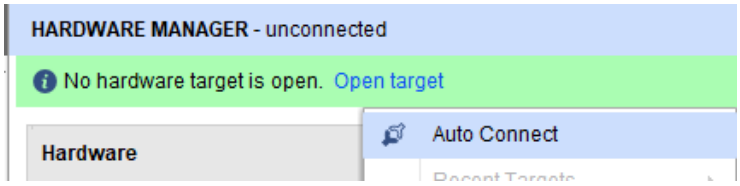

图 32 连接器件

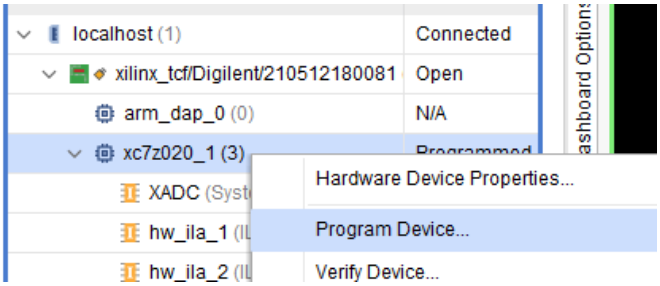

图 33 下载 bit 流文件

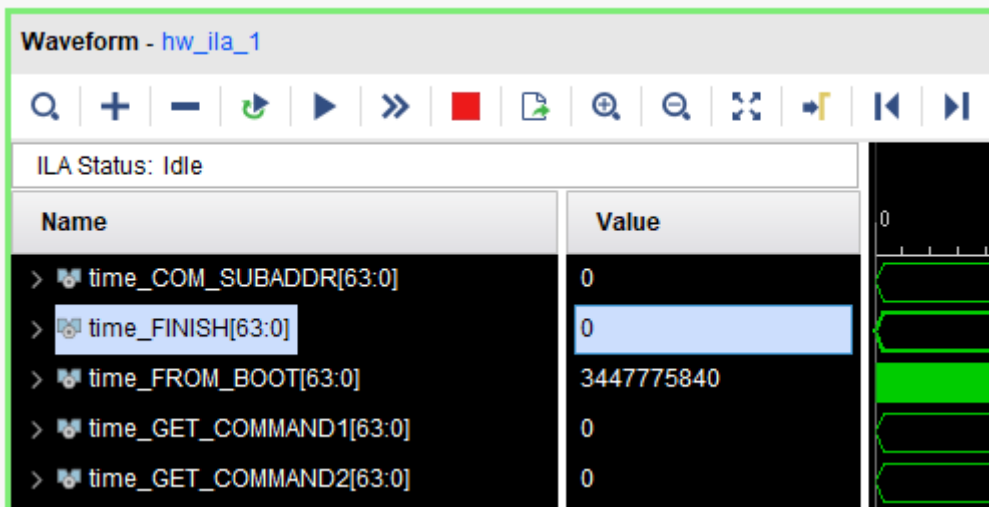

图 34 在 Waveform 中看到实时数据

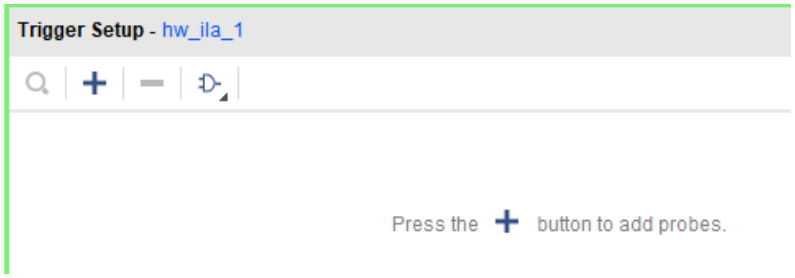

图 35 在 Trigger Setup 中设置触发条件

图 36 开始触发

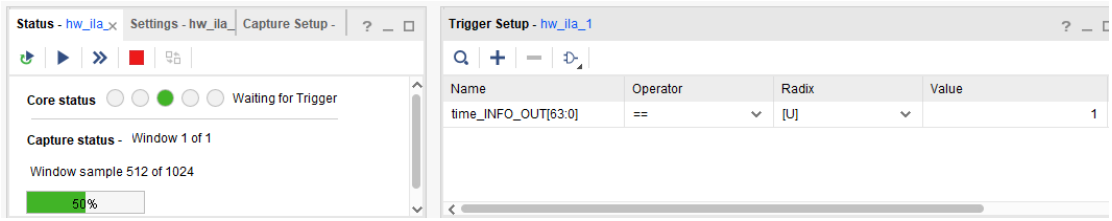

图 37 等待条件被触发到

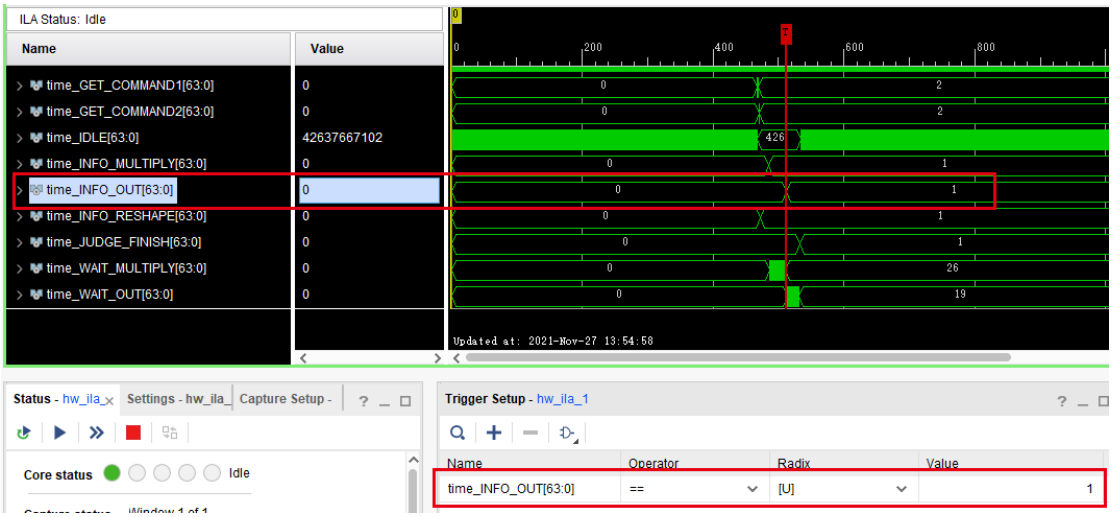

图 38 PS 送入数据后，在设定条件位置处触发

Lab4 中 Matmul.py 完成了矩阵输入写入 BRAM，用 pl_simulate.py 模拟了 FPGA 的操作。将 bit 下载后，无需执行 pl_simulate.py，执行 Matmul.py 即可。

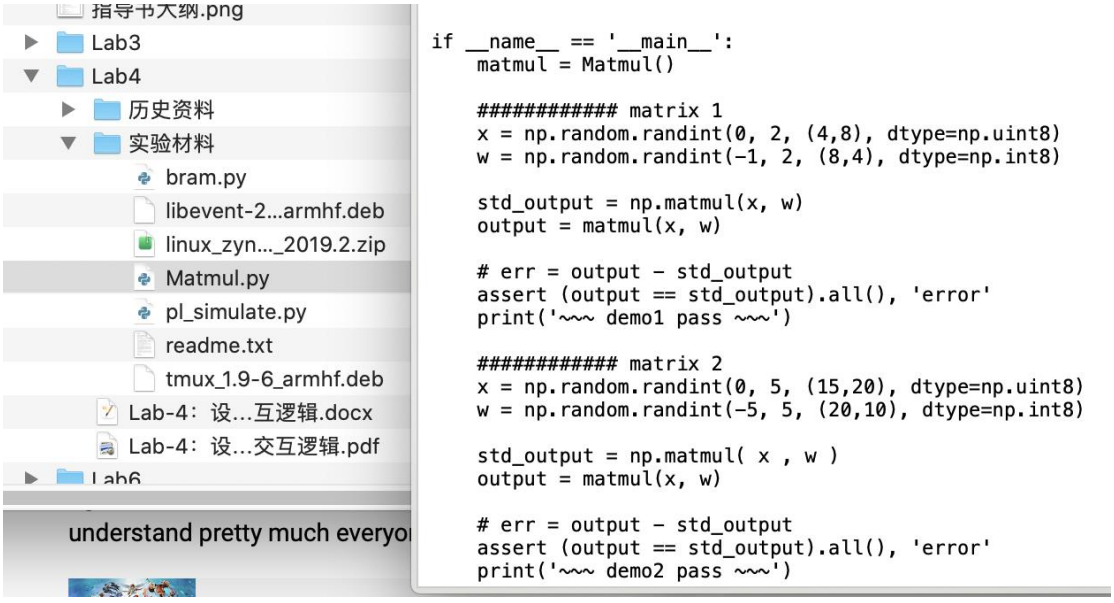

图 39 lab4 代码

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 ~~~
```

图 40 应打印结果

4.3 问题

本部分问题无需全部完成，但至少完成 **30※**。完成※越多，本次实验得分越高。

4.3.1 问题 1（3※）

依据工程中 BRAM_FM64 和 BRAM_WM128 的位宽、地址深度，写出 M、N、P 取值的限制。

4.3.2 问题 2（3※）

依据 mlp 和 lenet 模型，判断当前 BRAM_FM32、BRAM_WM32、BRAM_FM64 和 BRAM_WM128 大小是否足够。

4.3.3 问题 3（10※）

修改 MM_top_tb.v，利用$random 函数在 MM_top_tb.v 中产生随机的 M、N、P，依据产生的 M、N、P 来产生随机输入矩阵。之后，计算两个随机矩阵相乘的结果（三层 for 循环嵌套，注意不需要有时钟限制，仿真表现为立马算出结果），将结果暂存。将随机矩阵送入 MM_top 模块，待得到运算结果，和暂存的结果进行比较，打印出是否出错等必要信息。重复以上过程，不停地产生随机输入进行计算和结果比对，对 MM_top 模块进行充分的、随机化的验证。

4.3.4 问题 4（8※）

如图 2 矩阵乘法模块框图，输入矩阵首先写入 BRAM_FM_32b 和 BRAM_WM_32b，再转换写入 BRAM_FM_64b 和 BRAM_WM_128b。本教程提供工程示例，可直接下载该工程文件使用（指的是问答题工程文件压缩包），然后修改工程代码，将 PS 侧输入矩阵直接写进 BRAM_FM_64b 和 BRAM_WM_128b。提示，PS 侧和 PL 侧的交互不一定非要采用 AXI4LITE 协议，这意味着总线的数据位宽不一定非得是 32bit。

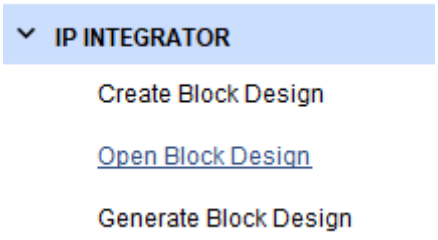

图 41 打开 Block Design

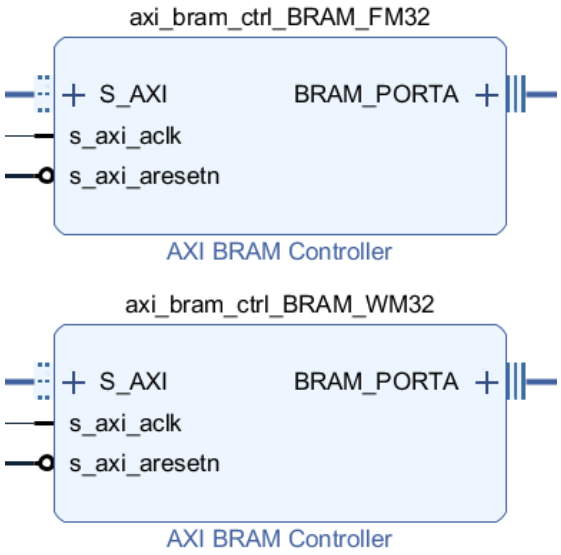

图 42 双击 AXI BRAM Controller 进行设置

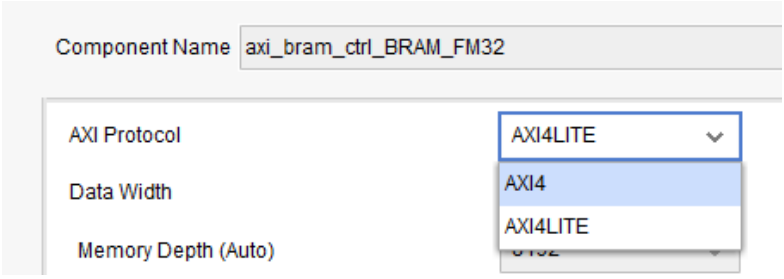

图 43 选择 AXI 协议

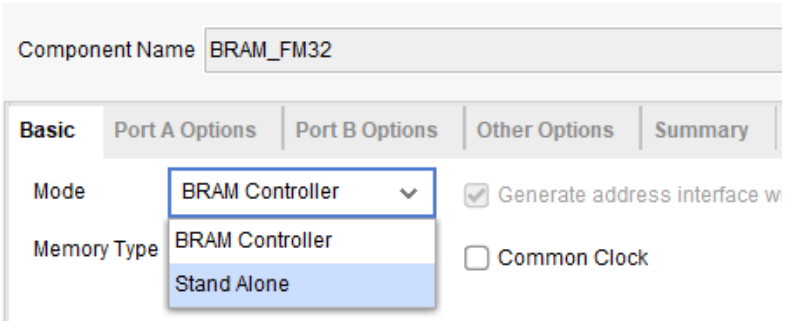

图 44 设置 BRAM 模式

（可以在 Stand Alone 模式下修改位宽后切换回 BRAM Controller 模式）

注意，输入矩阵直接写进 BRAM_FM_64b 和 BRAM_WM_128b 时，PS 侧补 0 的个数也需要进行修改。本题目另提供工程，无须对原有工程进行修改，修改 PS 侧补 0 代码，跑通流程即可。

4.3.5 问题 5（3※）

PS 侧输入矩阵会进行补 0 操作。从 PL 操作流程考虑，补 0 是否必要？如果不进行补 0，无效位置保留默认值，是否会影响矩阵运算结果？给出原因。

4.3.6 问题 6（8※）

Lab4 中实现了对 input、weight 补零后写入 BRAM 的 Matmul 类，补零操作存在一定时间开销。修改 Matmul 类实现，通过跳地址的方式将 input、weight 写入对应的 BRAM 块中。注意，有效数据存储位置相同，但跳地址的方式不进行补零，跳过本来应该填充零的位置。

4.3.7 问题 7（2※）

本工程中 FPGA 的时钟频率是多少？提示，在 Block Design 中双击 ZYNQ7 Processing System，点击 Clock Configuration 查看 PL Fabric Clocks 的时钟。

4.3.8 问题 8（5※）

仿照图 3 CTRL 模块状态转移图，画出 Multiply_ctrl 模块的状态转移图。并给出以下寄存器的功能：sub_scale_M1、sub_scale_P1、sub_scale_M2、sub_scale_P2。

4.3.9 问题 9（8※）

阅读 Align_fifo 模块代码，结合仿真波形描述数据写入和读出的过程。

4.3.10 问题 10（n※）

除了上述问题提到的，提出你认为设计上还存在改进的点，或提出你的完成本模块功能的方案。本题不设※上限，合理并有效的改进点计 5※，提出创新可行的方案思路计 12※，给出详细的方案计 25※。