实验六：CPU 的拓展与 Accelerator 的实现

在前五个 Lab 中，我们实现了五周期流水线的 RISCV 核和一个简易的 Cache 模块。在本次实验中，我们将实现一个基于脉动阵列的 Accelerator，并通过一个简易总线将 CPU、Instruction Memory、Main Memory、Accelerator 连接在一起。

1 实验目的

i. 掌握总线的基本运行机制；

ii. 掌握脉动阵列的实现方法；

iii. 通过对 CPU 指令集的扩展，了解 Accelerator 作为 co-processor 与 CPU 的协作方式；

2 实验环境（推荐）

与实验一相同：

i. IDE：vscode

ii. verilog compiler：iverilog

iii. waveform viewer：GTKwave

3 实验任务

本次实验要求实现一个基于脉动阵列的 Accelerator，并通过一个简易总线将 CPU、Instruction Memory、Main Memory、Accelerator 连接在一起。大家需要按照步骤完成以下三项任务。

(1) 实现一个简易 BUS（为降低难度，代码已经提供）。

(2) 实现一个基于脉动阵列的 Accelerator（部分控制代码已提供）。

(3) 拓展 RISC-V 核的 ISA、调整 BUS 上挂载的其他组分，完成对 Accelerator 的控制和协同。

3.1 BUS 的实现

Naïve bus 在整个系统中的定位如图1-(a) 所示。所有有关 BUS 本身实现的代码已经给出，请参考 naive_bus.v。因此在这一步骤只需要你阅读代码，了解 BUS 的原理，并后续在各个主从设备中调整相应信号以使用总线进行通信。

本 Lab 中的 Naive Bus 实现参考了 tinyriscv。大家可以参考链接中的硬件篇–3.9. 总线部分的内容来了解总线的相关知识与基本原理。

Naïve bus 分别使用 3bit、4bit 选择主、从设备，这意味着它最多支持 8 个主设备和 16 个从设备的通信。在本次实验中，RISC-V CPU 在取指阶段的指令存储访问接口和访存阶段的主存访问接口需要与总线的两个主接口相连；而指令存储模块、主存模块、以及大家后续需要实现的加速器模块需要连接到总线的从接口。总线同时支持一主一从通信，并且仲裁优先级固定（详见代码）。总线的数据宽

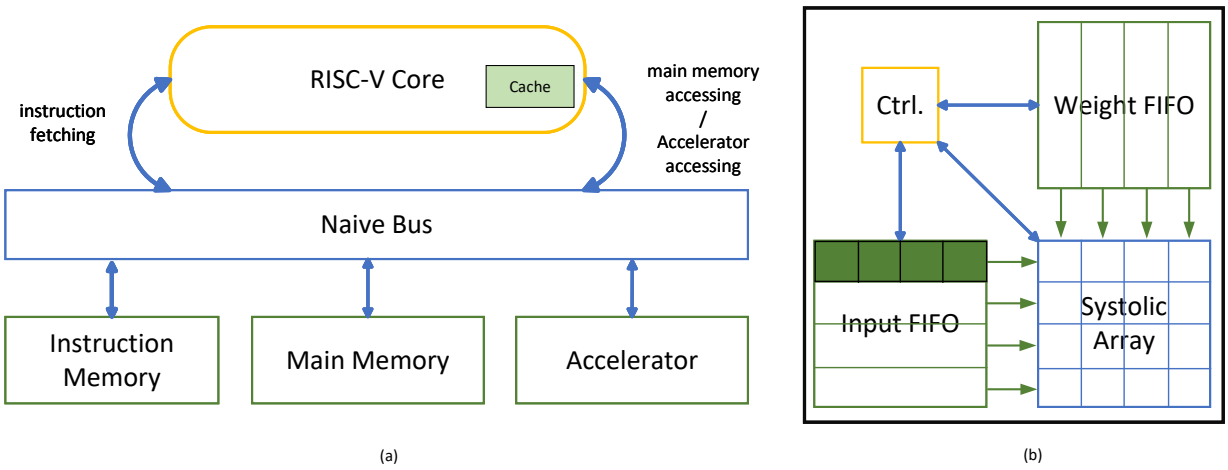

图 1: （a）总线与模块组织（b）Accelerator 结构

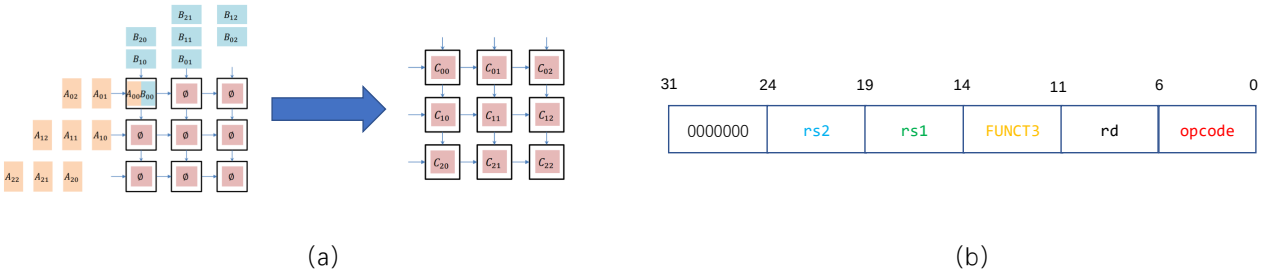

图 2: （a）脉动阵列数据流（b）指令结构

度被设置成 256bit——这刚好对应脉动阵列中一行/一列数据的大小，也刚好对应一行 Cache line 的大小（如下所述，在 Lab6 中，Cache Line 的长度会被扩大）。

在本次 Lab 中，考虑到 Systolic Array 连续的数据访问模式，为了充分发挥 Cache 的作用，我们扩充了 Cache 的容量——Cache 的参数 WAY_CNT 被调整为 16，并且使用 LRU 替换策略。

3.2 Accelerator

如图1-(b) 所示，Lab6 要实现的 Accelerator，由 Controller、Weight FIFO、Input FIFO 和 Systolic Array 四个主要部分组成。其中，Weight FIFO、Input FIFO 和 Systolic Array 是 Accelerator 的执行模块，会在本章介绍。Controller 是控制模块，在第4章与其他控制单元一同介绍。

Accelerator 在 systolic_accelerator.v 中实现，在该文件中，你需要将上述模块例化并连接。

在例化、赋值时，建议使用之前介绍的 generate、for 循环以简化代码、适应不同的 parameter。

3.2.1 PE

PE 是脉动阵列运算的基本单元，每个周期可以进行一次 MAC 运算。PE 模块实现在 pe.v 文件中，它包含一个参数：DATA_WIDTH=16，代表 PE 的数据位宽（默认为 16 位，也是本次实验中 Accelerator 的数据位宽，大家不需要改动）。PE 还包含五个输入端口（clk, rst, accumulate_enable, input_north, input_west）和三个输出端口（output_south, output_east, result）。

当 rst 信号被置为有效时，PE 模块会把输出信号全部清零。否则，在每个周期，每个 PE 从西、北两个方向接收传入的数据，根据控制信号 accumulate_enable 选择是否进行乘累加操作，并将部分和保存在其内的缓存寄存器中，同时将输入数据继续传向东、南两个方向。

大家可以直接使用 verilog 中的乘法运算符来实现 MAC 中的乘法运算。此外，为了实现下一节中介绍的 Systolic Array 数据流，大家也需要思考如何使用时序逻辑来实现 PE 模块。

3.2.2 Systolic Array

Systolic Array 是 Accelerator 的核心，它由 PE 阵列构成，在 Controller 的控制下，它将 FIFOBuffer 中的数据以如图2-(a) 的脉动的形式进行乘累加运算。请注意，Systolic Array 是 Output Stationary 的数据流，因此完成矩阵乘法后，矩阵乘的结果存储在每个 PE 内部的结果寄存器中。此外，Systolic array 的数据流中，两侧的输入数据中，相邻两行的数据输入需要间隔一个周期，以确保需要运算的数据被传到相同的 PE 上。

Systolic Array 模块实现在 systolic_array.v 文件中，它包含两个参数：Array_SIZE=16、DATA_WIDTH=16，分别代表阵列的尺寸和数据位宽。此外，它还包含七个输入（clk, rst, accumulate_enable, read_enable, west_inputs, north_inputs, row_index）和一个输出（results）。其中，west_inputs, north_inputs 分别对应 Input FIFO 和 Weight FIFO 输出的一列 Input/一行 Weight 数据。运算完毕后，Systolic Array 根据控制信号 read_enable 从阵列中读出行标号为 row_index 的一行数据，传给 Controller，由 Controller 将数据从 Accelerator 传递出去。

由于每个周期每个 PE 只会计算一次 MAC 操作，因此 Systolic Array 需要多个周期来计算完整的矩阵乘。我们为大家提供了一份名为 tb_systolic.v 的 test bench，其中对一个 4*4 的 Systolic Array 做了简单测试。大家可以参考其中的信号输入来理解 Systolic Array 的工作机制。这份 test bench 也会用来测试 Systolic Array 的实现正确性。

3.2.3 FIFO

考虑到 Systolic Array 每行/每列的顺序输入模式，我们使用 FIFO 来存储 Systolic Array 的输入。且为了简化设计，我们规定 Input FIFO/Weight FIFO 的总大小与 Systolic Array 的大小相同。这样对于一次运算，Input FIFO/Weight FIFO 刚好输出 buffer 内的全部数据。

FIFO 基本单元——FIFOBuffer 实现在 fifo.v 文件中。请注意，以 Input FIFO 第一行为例，如图1-(b) 所示，FIFOBuffer 模块仅对应 FIFO 阵列的“一条”，即 Input FIFO 阵列的一行或 Weight FIFO 阵列的一列。因此，大家需要在 systolic_array.v 中根据阵列的尺寸相应地例化多个 FIFO。

FIFOBuffer 模块有两个参数：DATA_WIDTH=16、BUFFER_SIZE=16，前者代表 FIFOBuffer 中每个寄存器的位宽，后者代表每个 FIFOBuffer 内的寄存器数量，它应该与前面提到的 ARRAY_SIZE 相等。FIFOBuffer 模块还包含了五个输入端口（clk, rst, inject, bubble, buffer_input）和一个输出端口（buffer_output）。其中，rst 信号负责控制 FIFOBuffer 内数据初始化（清零）；inject 信号负责控制 FIFOBuffer 将输入端口的 buffer_input 数据加载到 FIFObuffer 中（*为简化设计，我们假定 FIFOBuffer 可以在一个周期将所有输入数据加载进来，因此 buffer_input 宽度与整个 FIFOBuffer 容量相等。*）；bubble 信号负责控制 FIFOBuffer 向外弹出当前 Buffer 内的首个数据。与此同时，其他数据也需要整体向端口方向移动一个单位。（*FIFOBuffer 每个周期至多弹出一个数据，因此 buffer_output 宽度与 DATA_WIDTH 相等。*）

4 各组分的连接和整合

Accelerator 作为协处理器，它根据 CPU 的相关控制信号来执行不同的操作。因此，我们需要对 CPU 进行 ISA 扩展，以实现控制 Accelerator 的相关指令，并且根据这些指令生成相应的控制信号。因此，在本部分，你需要完成以下三部分工作：

(1) CPU 的 ISA 扩展：包括流水线上各个信号的生成与传递；

(2) Accelerator Controller 的实现：Accelerator 作为协处理器，无法自行取指译码访存运算，因此 Controller 需要接收 CPU 相关信号，对 Accelerator 中的另外三个组分进行控制和操作；

(3) CPU 和 Cache 配合总线和 Accelerator 需要进行进一步改动；

4.1 ISA 扩展

我们共扩展了五条指令，指令结构如图2-(b) 所示。每条指令的结构、功能，以及我们给出的信号交互的参考格式如下所述。特别地，所有指令的 opcode 字段均为 1111111。

(1) LOAD：rs1：memory addr, rs2：accelerator addr, funct3：000

这条指令负责将输入数据从内存加载到 Accelerator 的 FIFO 中。请注意，这条指令以 FIFOBuffer 的大小为粒度进行数据搬运，并且大家也可以假设 FIFOBuffer 内的数据是连续存放在内存中的。

为了实现这条指令，CPU 需要执行的操作为：读取缓存（本质上是读取内存，可能命中也可能缺失）；然后在读到有效数据的当周期内将数据加载到总线上，并传给 Accelerator（如前所述，我们可以假设 Accelerator 的数据读写当周期内就能完成）。

为了控制 Accelerator，在传输数据的同时，我们规定 CPU 需要使能总线上的 write_request 信号，并且传入对应 FIFOBuffer 的起始地址（地址映射详见后文）。为了简化实现，大家可以假设程序中传入的地址与每个 FIFOBuffer 的起始地址对齐。此外，此指令只操作 FIFO 缓冲区，因为只有 FIFO 需要接收外部的数据输入。

(2) SAVE：rs1：memory addr, rs2：accelerator addr, funct3：001

这条指令负责将 Systolic Array 的结果矩阵从 Accelerator 写入内存中。与 LOAD 指令类似，这条指令以输出矩阵的一行为粒度进行数据搬运，这也与我们在前述 Accelerator 的实现中的数据搬运粒度相对应。

为了实现这条指令，CPU 需要执行的操作为：首先通过总线从 Accelerator 中读取数据（当周期内获取），然后写入缓存（执行 Cache 操作）。

为了控制 Accelerator，在传输数据的同时，我们规定 CPU 同时需要使能 read_request 信号，并且传入对应输出行的起始地址。此外，这条指令对输出缓冲区（脉动阵列中的寄存器）进行操作，因为只有结果寄存器需要向内存传输输出。

(3) MATMUL：funct3：010

这条指令负责向 Systolic Array 传输矩阵乘请求。它会在所有的数据都被加载进 FIFO 阵列之后调用，因此 CPU 不需要向 Accelerator 传递数据信息。为了控制 Accelerator，CPU 需要使能 write_request 信号，然后将 Accelerator 访存地址设置为输出缓冲区的初始地址（相当于通知 Accelerator 需要计算结果并将其写入到输出 Buffer 中）。

Accelerator 收到相应的信号后，Controller 会控制 FIFO 向外输出数据，同时 Systolic Array 以脉动方式进行运算。如前所述，矩阵乘法需要 Accelerator 花费多个周期完成，在 Accelerator 执行期间，CPU 需要保持信号，并且暂停流水线。

(4) RESET：rs2：accelerator addr, funct3：011

这条指令负责向 Accelerator 传输数据清零请求。Accelerator 根据传入地址的所在区域（Input/Weight FIFO，Output Buffer）清零对应的区域内容，该指令传入的地址为每片存储区域的起始地址。

为了控制 Accelerator，我们规定 CPU 需要同时使能 read & write 信号，并且将目标缓冲区的初始地址送入总线。

(5) MOVE：funct3：100

这条指令实现了加速器的片上数据复用。它将 Systolic Array 的所有输出逐行转移到 Accelerator 的 Input FIFO 阵列中。

为了控制 Accelerator，我们规定 CPU 需要同时使能读写信号，然后将 Accelerator 访存地址设置为 MOVE 对应特殊地址，从而与 Reset 指令进行区分。

该指令 Accelerator 需要多个周期完成（因为阵列中有多行），因此与 Matmul 指令类似，在 Accelerator 执行期间 CPU 需要保持信号。

由于 Input FIFO 大小与 Output Buffer 大小相等，在我们提供的两层全连接层网络的测试代码中，上一层的输出作为下一层的输入，因此可以直接将 Output Buffer 中的数据拷贝到 Input FIFO 中。

大家可以查看本次 Lab 中新增的三份测试代码来进一步熟悉上述 ISA 的用法。

上述 ISA 拓展中涉及到了总线上的全局地址（memory addr）和加速器内的局部地址（accelerator addr），其划分规则如下所示。

```
// definitions for base addresses of bus slaves
`define INSTR_MEM_BASE_ADDR  32'h00000000
`define DATA_MEM_BASE_ADDR   32'h10000000
`define ACCELERATOR_MEM_BASE_ADDR 32'h20000000

// definitions for systolic array accelerator
`define INPUT_FIFO_BASE_ADDR 32'h00000000
`define WEIGHT_FIFO_BASE_ADDR 32'h00000200
`define OUTPUT_BUFFER_BASE_ADDR 32'h00000400
`define MOVE_ADDR           32'h00000600
```

对于全局地址，受 instmem、datamem、accelerator 总内存大小的限制，常规寻址不会用到高 4 个 bit，因此我们用高位的 4 bit 选择 slave。所以 CPU 在传输地址到总线时，也需要考虑总线 Slave 的划分带来的基地址偏移。

对于加速器内的局部地址，我们规定从地址低到高依次为 Input FIFO、Weight FIFO、Output Buffer。特别地，两个 FIFO 阵列的编址均保证每个 FIFOBuffer 内部的地址是连续的。因此，从矩阵乘的角度来看（Input 矩阵乘 Weight 矩阵），Weight 矩阵的数据编址为转置之后再逐行存储。为了简化实现，我们在提供给大家的测试代码中也使用这种方式存储数据，因此大家不需要在硬件实现上额外考虑转置问题，只需要考虑以 FIFOBuffer 模块为粒度的数据填充。

4.2 Controller 实现

Controller 是整个 Accelerator 的控制单元，负责 Accelerator 与 BUS 的数据通信、控制 FIFO 和 Systolic Array 行为、转发数据给 Weight/Input FIFO、收集脉动阵列结果返回到 BUS。

Controller 共设置四个参数，其中 DATA_WIDTH=16，对应 LOAD 的数据、FIFO 中的数据、运算过程中的数据的位宽；ARRAY_SIZE=16，脉动阵列的维度；FIFO_BUFFER_SIZE，FIFO 的 Buffer 宽度，与脉动阵列的维度相同；PACKET_WIDTH=DATA_WIDTH×ARRAY_SIZE=256，对应从总线打包输入/输出的数据总位宽。在3.1节提到，为简化传输，我们让总线位宽刚好与之对应。

考虑到降低实验难度，我们给出了这部分的代码，详见 *controller.v*。我们不强制大家使用这部分代码，但大家需要保证最后能够正确地完成上述 *ISA* 拓展中的所有指令。

4.2.1 “指令译码”——根据 slave 信号判断操作

```
// judge which request has been sent to accelerator
wire load_request = ((!bus_slave_read_request && bus_slave_write_request) && (bus_slave_addr <
    `OUTPUT_BUFFER_BASE_ADDR));
wire save_request = (bus_slave_read_request && !bus_slave_write_request);
wire matmul_request = ((!bus_slave_read_request && bus_slave_write_request) && (bus_slave_addr >=
    `OUTPUT_BUFFER_BASE_ADDR));
wire reset_request = ((bus_slave_read_request && bus_slave_write_request) && (bus_slave_addr <
    `MOVE_ADDR));
wire move_request = ((bus_slave_read_request && bus_slave_write_request) && (bus_slave_addr ==
    `MOVE_ADDR));
```

如上述代码所示，根据读写请求信号和访问地址的范围判断当前 Accelerator 需要进行的操作。slave 信号为 master（CPU）传输得到，这部分的判断逻辑与 ISA 拓展中给出的 CPU 信号传递规则是呼应的。

如前所述，上述判断规则仅供参考，大家可以自行定义译码规则，并在报告中给出。

4.2.2 内置的有限状态机

```
// controller state machine
always @(posedge clk) begin
    if(rst) begin
        // init registers
        state <= READY;
        matmul_finish <= 1'b0; move_finish <= 1'b0;
        tmp_row_idx <= 0; tmp_matmul_cycle <= 0;
    end

    else begin
        case (state)
            READY: begin
                if(matmul_request) begin
                    // turn the state to Matmul
                end

                else if(move_request) begin
                    // turn the state to Move
                end
            end
```

```
            MATMUL: begin
                // Systolic till last element
            end

            MOVE: begin
                // Move till last line
            end
        endcase
    end
end
```

如上述代码所示，对于 Accelerator 来说，Load、Store、Reset 操作可以在当周期内完成，而 Matmul 和 Move 操作需要多个周期才能完成，因此为这两个指令的执行设计了一个有限状态机控制行为。

4.2.3 请求完成信号的产生

```
// judge whether to send finish signal to the CPU
assign bus_slave_request_finish = load_request ? (state==READY) :
                                  save_request ? (state==READY) :
                                  matmul_request ? ((state==READY && matmul_finish)) :
                                  reset_request ? (state==READY) :
                                  move_request ? ((state==READY) && move_finish) :
                                  1'b0;
```

如上述代码所示，与4.2.2节同理，Matmul 和 Move 操作需要多个周期才能完成，因此需要增加对 finish 信号的判断。再次强调，在 Accelerator 执行多周期操作时，CPU 应该暂停流水线，等待 Accelerator 的返回。

4.2.4 Systolic Array 与 FIFO 的数据、控制信号传递

我们在 Controller 内使用组合逻辑来传递数据和控制信号。由于篇幅限制，我们略去了对这部分代码的分析，请大家自行查看。如果大家选择使用我们提供的代码，只要大家把 Controller 到 Accelerator 其他模块的接口进行正确的连接，就可以保证功能的正确性。

4.3 CPU&Cache 的改动

4.3.1 CPU 流水线上的改动

一方面，在译码、执行等阶段，CPU 需要修改相应的逻辑以传递上述 ISA 拓展所需的关键信号；另一方面，流水线暂停的相关逻辑也需要修改，以保证 Accelerator 执行多周期操作时的信号正确性。

4.3.2 CPU MEM 模块的改动

如图1-(a) 所示，RISCV Core 与 Naïve bus 通过两个接口进行交互：第一个接口负责取指令（通过 BUS 访问 InstMem），而第二个接口同时负责对 Main Memory 的访问和与 Accelerator 的交互。这

样设计的原因是真实硬件中的引脚数量往往是有限制的，我们不能为每个新添加的模块都分配专用的引脚。这也是为什么我们要使用总线结构来连接不同模块的原因之一。

为了实现的简便性，考虑到我们是在 MEM 阶段进行内存访问，我们将与 Accelerator 交互的数据生成也放到同一个阶段进行实现。因此，CPU 的数据总线接口接受的数据需要在 CPU 向 Accelerator 传输的总线数据与原有的 Cache（在发生缺失或写回脏数据时）向 Memory 传输的总线数据之间进行多选。多选的逻辑大致如下所示：

```
// judge whether to take up the bus for the accelerator
wire accelerator_take_up_bus = (accelerator_instr_mem && (instr_funct3_mem==`LOAD ||
    instr_funct3_mem==`SAVE)) ? (cache_request_finish && !accelerator_request_finish) :
                           (accelerator_instr_mem && (instr_funct3_mem==`MATMUL || instr_funct3_mem
                               ==`RESET || instr_funct3_mem==`MOVE)) ? (!accelerator_request_finish
                               ) :
                           1'b0;
// muxing between accelerator and cache
assign data_bus_read_request = accelerator_take_up_bus ? accelerator_bus_read_request :
    mem_read_request;
assign data_bus_write_request = accelerator_take_up_bus ? accelerator_bus_write_request :
    mem_write_request;
assign data_bus_write_data = accelerator_take_up_bus ? accelerator_bus_write_data : mem_write_data
    ;
assign data_bus_addr = accelerator_take_up_bus ? accelerator_bus_addr : mem_addr;
```

特别地，考虑到部分指令在 Accelerator 侧需要多个周期才能完成，因此我们建议大家额外添加一个状态寄存器 accelerator_request_finish，来记录完成的状态。此外，大家也需要注意，LOAD 和 SAVE 指令需要先后对 Memory（Cache 缺失时）和 Accelerator 进行访问，因此在判断这两条指令在 Memory 和 Accelerator 间的总线仲裁也需要特殊处理。Accelerator 执行完毕状态的更新逻辑与生成 CPU 向 Accelerator 发送数据的相关逻辑的实现留给大家思考。

4.3.3 Cache 的改动

Cache 需要做的改动主要有两方面：一方面，Cache 生成的访存地址需要按照前述的总线地址分配进行修改；另一方面，由于 LOAD/SAVE 指令需要做 Accelerator 与 Cache 间的数据读写，因此大家需要额外添加传输读写数据数据的通路，并且添加选择信号来判断当前传入 Cache 的请求是常规的访存请求还是与 Accelerator 的数据交互请求，并且根据请求类别进行相应的数据读写操作。

4.4 模块组织

与 Lab5 类似，由于大家的 CPU 实现方式、Cache 可能与我们给出的组织方式有不同，因此本次实验中我们只为大家提供总线相关代码（naive_bus.v）、主存相关代码（main_memopry_wrapper.v 和 ram.v）、指令存储相关代码（instr_mem.v）、顶层模块相关代码（riscv_top.v），以及 Accelerator 的 Controller 相关代码（accelerator_components\controller.v）。

请大家在维持顶层模块的接口定义不变的前提下进行进一步的修改与整合，修改顶层模块的接口可能导致无法取得成绩。

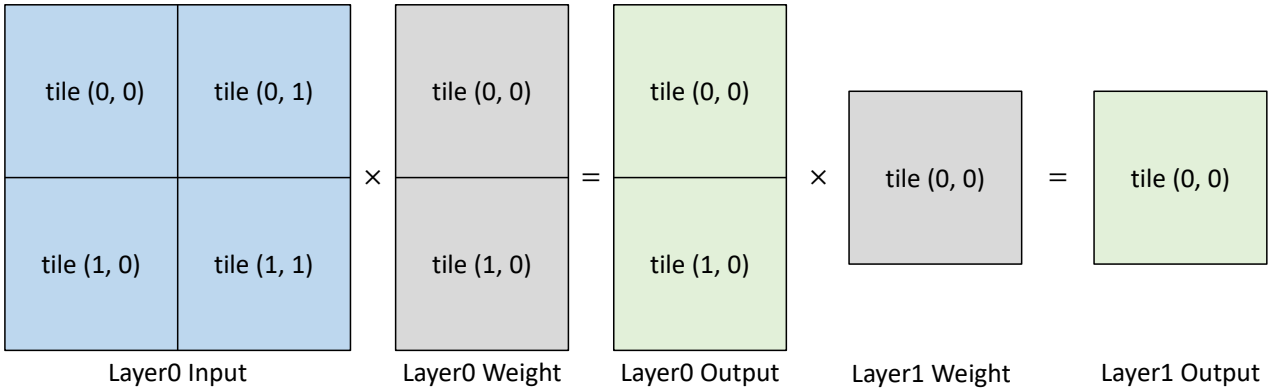

图 3: 2_layer_fc(_flush).s 中的两层全连接网络的矩阵乘示意图

4.5 信号设计

本次实验与 Lab5 类似，请大家根据自己的想法与前述各个模块的功能和描述，自行添加与修改信号。

5 测试与评分

请在本地调试完毕后，再将代码提交到测试平台，并查看测试平台中显示的分数。

5.1 tb 验证思路

本次实验中，我们使用 tb 文件进行 Accelerator 实现的正确性测试。其中，tb_systolic 提供了一个简单的 4*4 脉动阵列测试，用于你确认自己 FIFO、PE、systolic_array 设计的正确性，通过该测试你将获得 20 分。而针对将 Accelerator 实现并与总线相连的 SOC，使用 tb_soc 进行测试。共为大家提供了 5 份测试代码，每份代码的测试中对比 mem、cache 和 Systolic Array 的结果，都正确获得 16 分，满分 80 分。其中，2 份为 Lab5 中使用过的快速排序（quick_sort），这两份代码用于测试 Bus 实现与链接的正确性。其余三份测试 Accelerator 实现的正确性、ISA 扩展的正确性。

Accelerator 正确性测试的三份代码中，basic_functionality_test 用于测试五条拓展指令的功能正确性，在汇编代码中包含了所有指令的格式、使用方法、含义等信息。而另外两份代码则实现了一个简单的两层的全连接层网络（两次连续的矩阵乘），如图 3所示。

具体来说，网络的第一层为 32*32 的输入矩阵与 32*16 的权重矩阵相乘，得到 32*16 的输出矩阵；第二层用第一层的输出矩阵作为输入，与 16*16 的权重矩阵相乘，得到 32*16 的输出矩阵。由于 Systolic Array 每次只能计算 16*16 的矩阵乘法，因此我们在代码中设计了基于分块矩阵乘的数据流。代码首先计算第一层输入中 tile(0,0) 与权重 (0,0) 的乘积，然后计算 tile(0,1) 与权重 (1,0) 的乘积，两个部分和相加，得到第一层输出中的 tile(0,0)。然后代码会利用片上搬运指令，将其搬运到 Input FIFO 上，然后加载第二层的权重，二者进行运算得到第二层输出的 tile(0,0)。类似地，代码会重复上述流程，进行下半部分结果的计算。

5.2 重要说明

请确保提交时注释掉自己的 debug 代码，输出大量额外信息会给服务器造成很大压力。

随着验证逻辑越来越复杂，线上平台的评测压力较大，也更容易出现网络波动导致出现“运行时错误”的情况。当你发现自己本地跑通的代码报了运行时错误，且在报错界面没有显示 ERROR 信息时，可以联系助教，或等待一段时间后提交。

如果发生“等待评分”状态，请刷新界面，如果仍然是“等待评分”，切勿重复提交，请联系助教。

5.3 平台使用

1. 课程平台地址：微处理器设计与智能芯片课程实践
2. 请按规定字段进行注册，并牢记自己的密码。
3. 从测试平台上下载的代码包包括模板文件和简化的测试文件，请先仔细阅读 README，其中提到了你至少应该上传哪些文件，可以自行定义其他辅助模块一并压缩上传。
4. 测试平台会用自己的测试用例覆盖提供给同学们的用例，可以不再上传与测试相关的文件。
5. 请珍惜实验室服务器资源，切勿恶意提交代码。
6. 与平台、实验有关的任何问题可以与助教联系。
7. 欢迎大家提出有关平台、实验的各种建议，让本课程和配套实习越来越好。

5.4 评分规则

tb_systolic 对应 20 分。tb_soc 对应 5 份测试代码，每份 16 分，共 80 分。

注：平台评分主要作为对你完成度的提示，并不等于你最终课程实习部分的分数。

5.5 实验报告

在实验的同时和实验之后撰写实验报告是非常良好的习惯。这不仅有助于提高实验效率，理清实验思路，帮助 debug，也有助于记录实验过程和结果，还可以和其他同学交流和分享。因此，我们需要大家在 lab 完成过后为我们提交一个实验报告，实验报告有以下几点注意事项：

1. 包含你的设计思路，可以包括相应的模块组织层次、一些不同于助教提示的地方等。
2. 你 debug 的过程，或崩溃或大起大落或柳暗花明的心路历程等。
3. 最终用于 Lab 评分的那次提交：包括代码包、分数截图、提交时平台为本次提交生成的 ID。
4. 实验报告不做字数要求，体现思考和实验过程即可（反卷第一名）。即使最后有 bug 没解决，没有以满分通过平台测试，讲一讲你的 debug 过程与思考也会获得一些分数的补偿:）

6 提示与帮助

会在课程网站上动态更新，大家遇到问题先去网站的 Q&A 板块看一下。