SIMD NPU 设计文档

郭睿康 22307130115

2025 年 6 月 19 日

1 设计概述

本次实验设计了一款 SIMD 的向量处理 NPU，旨在通过数据级并行的方式加速矩阵乘加运算。该设计支持 8*8 规模以下的 16 位定点矩阵的乘加运算加速，并可以自行配置矩阵的规模进行计算。在指令集架构层面，该设计与 Slide 中的指令集架构保持一致，在标量指令集的基础上做了向量指令拓展；在硬件架构层面，该设计也与 Slide 类似，采用了 4 级流水的 RISC 结构，同时设置了可以同时读写的标量和向量寄存器堆和存储器，但在实现细节上略有差异，体现在以下几个方面：

1. 在 MOV 指令的实现通路上，添加了一级寄存器以保证所有指令都能够时序对齐，避免了 RAR 或 WAR 数据依赖或结构冲突；
2. 在运算过程中，除了计算阶段，该系统还设计了数据初始化 (数据和指令 setup) 阶段和数据输出阶段，在没有板子的情况下也可以实现高效的仿真验证；
3. 完善了电路的数据前馈路径，同时将寄存器设置为下降沿写，上升沿读，有效处理了各种指令序列中的数据依赖问题，一定程度上提升了系统的通用性；
4. 对张量存储器，向量存储器和指令存储器做了统一编址，实现了类似于 AXI 的简易接口逻辑，用于指令和数据高效的加载和读出，便于系统的仿真验证；

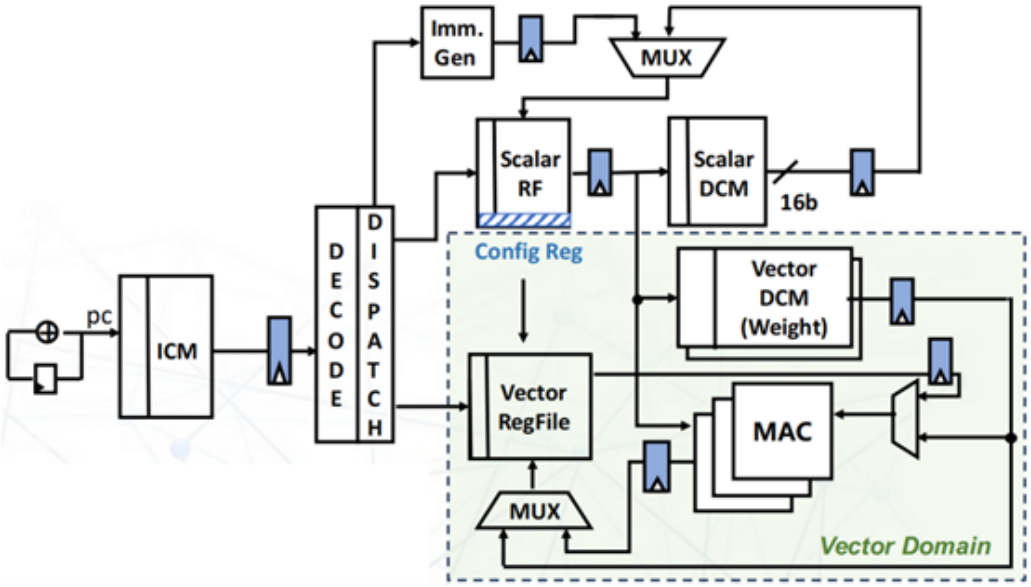

图 1: 硬件架构

软件架构层面，设计了 MATLAB 脚本用于生成测试指令序列，该脚本可以根据用户指定的矩阵规模自动生成汇编指令序列，并产生对应的机器码和 testbench，可以方便地进行测试和验证。由于整体上架构和 Slide 中介绍的架构基本相同，下文中只介绍与 Slide 不同部分的细节。

1.1 指令集架构

该设计的寄存器和总线位宽为 16 位 (数据位宽、地址位宽和指令位宽都是 16 位)，总的可用的地址空间大小为 128KB(在该设计中，一个地址就对应 16 位的数据)，然而这么大的空间会耗尽板子上的存储资源，因此只使用其中很小的一部分。统一编址情况如下：

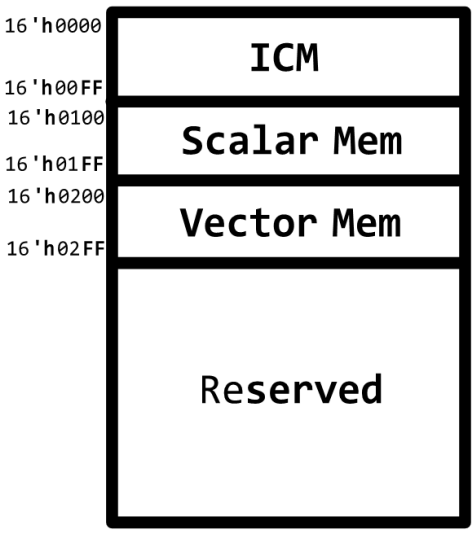

图 2: 地址空间分配

该处理机可以接收外部地址，控制器根据地址所在范围将地址自动映射到对应的存储器上进行读写。寄存器方面，总共设置了 16 个向量寄存器 (每个向量寄存器大小为 128 位，即可以存储 8*16 位的定点数) 和 16 个标量寄存器 (寄存器位宽也为 16 位)。编号为 0 的寄存器硬布线到 0，编号为 15 的张量寄存器低 8 位为 vlen，也即当前矩阵规模；高 8 位为 vmask，也即当前矩阵的掩码。其余寄存器均可正常使用。

指令方面，该处理机支持 6 种指令，三条标量指令和三条向量指令，指令格式如下所示：

表 1: 指令格式

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOAD	000			rd 4b				rs1				imm5b				
Store	001							rs1				rs2				
MOV	010			rd 4b				imm 8b								func 1b
Vload	100			vrd 4b				rs1				imm5b				
Vstore	101							rs1				vrs				
VMAC-s	110			vrd 4b				rs1				vrs				func 1b

指令的功能和 Slide 中所叙述的完全一致，此处不再一一赘述。值得注意的是，这个指令集的设计并不好，LOAD 和 Vload 中 5 位的立即数不够 8*8 规模的矩阵进行寻址。

1.2 硬件架构

硬件架构方面，有如上的四个主要的细节，下面一一阐述。

1.2.1 MOV 指令的实现通路

从原来的架构来看，除了 MOV 指令，其他所有指令都需要 4 个周期完成，而 MOV 只需要 3 个周期，会提前 1 个周期写回寄存器。这可能造成不同指令写回阶段的重叠，造成结构冲突，因此在具体实现上，虽然 MOV 的第三拍是空拍，但还是让它打一拍之后再写回，也就是说要在如下图所示，在立即数写回之前加一个寄存器。

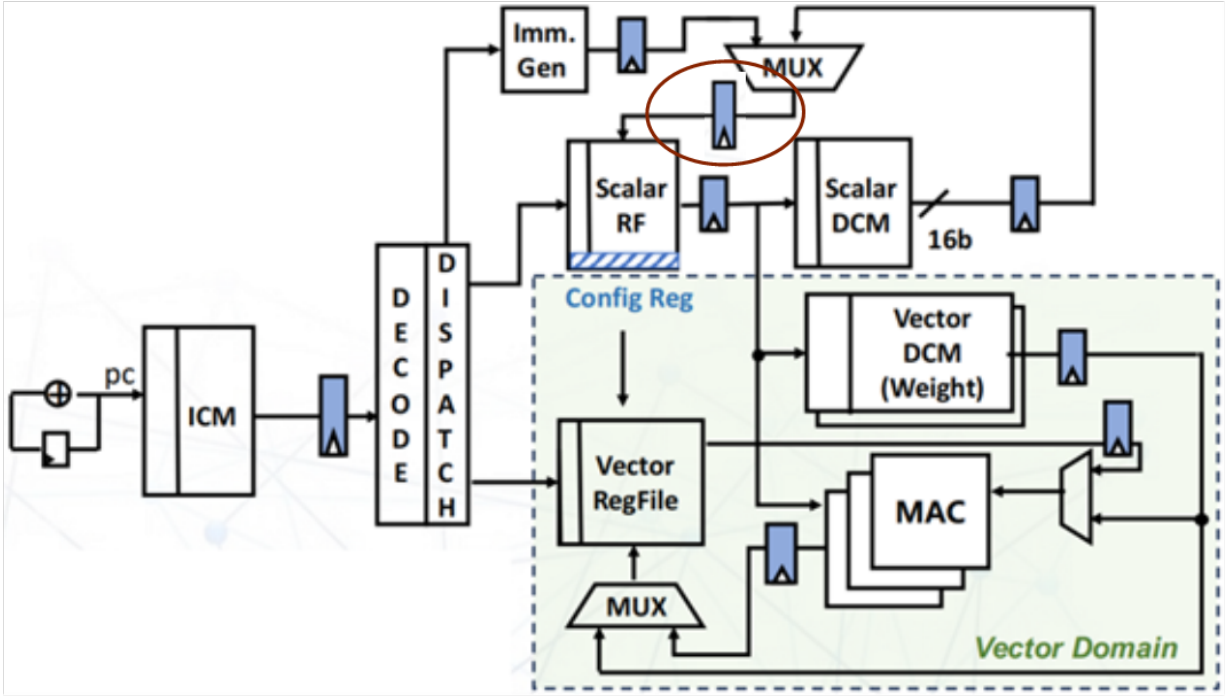

图 3: 改进后的架构

具体代码细节无需多言。

1.2.2 接口逻辑

为了简化仿真验证，方便地载入指令数据和读出结果，在处理机的输入和输出侧设计了简易类似于 AXI 接口逻辑，实现与片外的通信，这里详细介绍数据输入端，输出端是类似的。如下所示，展示了输入端的时序图：

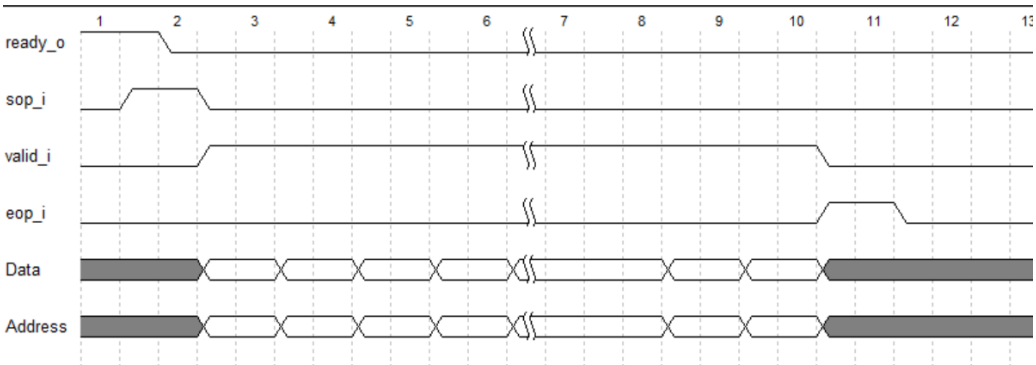

图 4: 简易 AXI 接口逻辑

当处理机端准备好接收数据（Ready_o 拉高），外部（CPU）会可以拉高 sop_i 表示数据传输开始，下一拍开始传输数据 (包括指令) 和地址，片上内部控制器将指令映射到正确的物理存储器上，将数据写入存储器或从存储器中读出。地址映射部分代码如下所示：

```
1         always @(*)
2         begin
3             if (state == INIT && valid_i) begin
4                 if(addr_in_i < SCALAR_BASE) begin
5                     wen_init_o = 3'b001;
6                 end
```

```
                else if (addr_in_i < VECTOR_BASE) begin
                    wen_init_o = 3'b010;
                end
                else if (addr_in_i < VECTOR_UPPER) begin
                    wen_init_o = 3'b100;
                end
                else begin
                    wen_init_o = 3'b000;
                end
            end
            else begin
                wen_init_o = 3'b000;
            end
        end
```

wen_init_o 用于控制初始化阶段向存储器写入数据，是三个存储器的使能信号。

1.3 有限状态机设计

该处理机的工作流包括五个状态：空闲 IDLE、数据和指令初始化 INIT、矩阵计算 COMP、计算完成 DONE、正在写回 TRAN。COMP 状态完成处理机的核心工作，而另外几个状态则主要用于仿真验证，状态转移的代码如下所示：

```
        always @(*)
        begin
            case (state)
                IDLE: begin
                    if (ready_o && sop_i) begin
                        next_state = INIT;
                    end
                    else begin
                        next_state = IDLE;
                    end
                end

                INIT: begin
                    if (eop_i) begin
                        next_state = COMP;
                    end
                    else begin
                        next_state = INIT;
                    end
                end

```

```
                COMP: begin
                    if (complete_o) begin
                        next_state = DONE;
                    end
                    else begin
                        next_state = COMP;
                    end
                end

                DONE: begin
                    if (result_is_OK_o && ready_i)
                        next_state = TRAN;
                    else begin
                        next_state = DONE;
                    end
                end

                TRAN: begin
                    if (trans_complete_o)
                        next_state = IDLE;
                    else
                        next_state = TRAN;
                end

                default: begin
                    next_state = IDLE;
                end
            endcase
        end
```

具体细节不赘述，其中的 complete_o,result_is_OK_o,trans_complete_o 均是由计数器驱动的控制信号，分别用于表征计算完毕、结果已经全部写回向量存储器、结果数据输出完毕。

1.4 数据前馈系统

原设计中，只考虑了 Vload 后接 VMAC-s 指令这一种数据前馈情况，一是没有考虑隔一条的指令之间的数据冒险，二是没有考虑其他很多指令序列可能存在的 RAW 问题（虽然这样的指令序列不一定会出现）。为了解决这两个问题，采用了更加丰富的数据前递策略。将两类寄存器都设置为下降沿写上升沿读，避免了相隔一条指令的两条指令间的 RAW 问题，在指令译码阶段就能把数据准备好；对于第二个问题，根据指令序列情况设计标志位，对写回数据，写回地址以及 MAC 单元的操作数进行选择，用传统方法实现数据前递，解决数据还没写回寄存器但已经需要被读的问题。例如如图所示的结构：

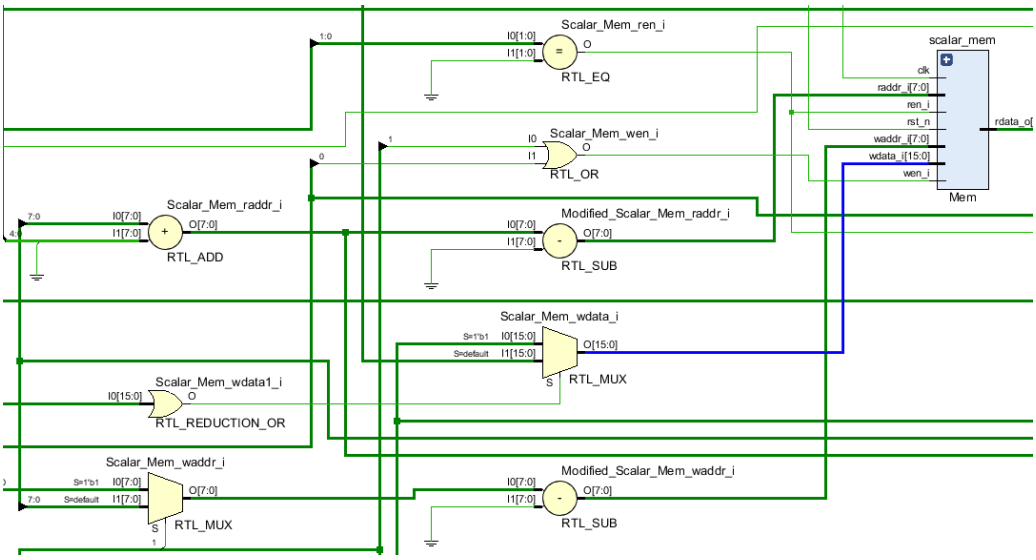

图 5: 数据前递结构示例——标量存储器的地址和数据的数据前递

```
        MOV r1 02 $1
        MOV r1 00 $0          % Load Base Address
        ...
        MOV r15 8 $0
        MOV r15 FF $1         % Config Register
        Vload vr1 r1 0
        VMAC-s vr2 / vr1 $1   % Load Bias
        Load r5 r3 0          % Load Input
        Vload vr1 r2 0        % Load Weight
        VMAC-s vr2 r5 vr1 $0 % MAC Calculation
       ...
        VStore / vr2, r4      % Store Result (wo/ offset)
        MOV r4 02 $1          % Modified result Address
        MOV r4 88 $0          % Modified result Address
```

1.5 软件架构

软件部分直接使用汇编语言编写，并用 MATLAB 实现了汇编语言的生成和编译脚本，高效完成指令序列的生成和仿真平台的搭建。算法层面采用和 Slide 中完全一致的策略，将第一个矩阵的元素与第二个矩阵的对应行做向量数乘累加得到结果。略有不同的是，在计算 8*8 矩阵时，立即数不够用，而又没有实现加法指令，因此要事先产生两个基址，且由于地址是 16 位的，需要两条 MOV 指令才能完成基址的加载。报告中贴出了汇编代码片段，供参考。

1.6 系统 setup 与验证流程

系统的 setup 和验证流程如图所示：

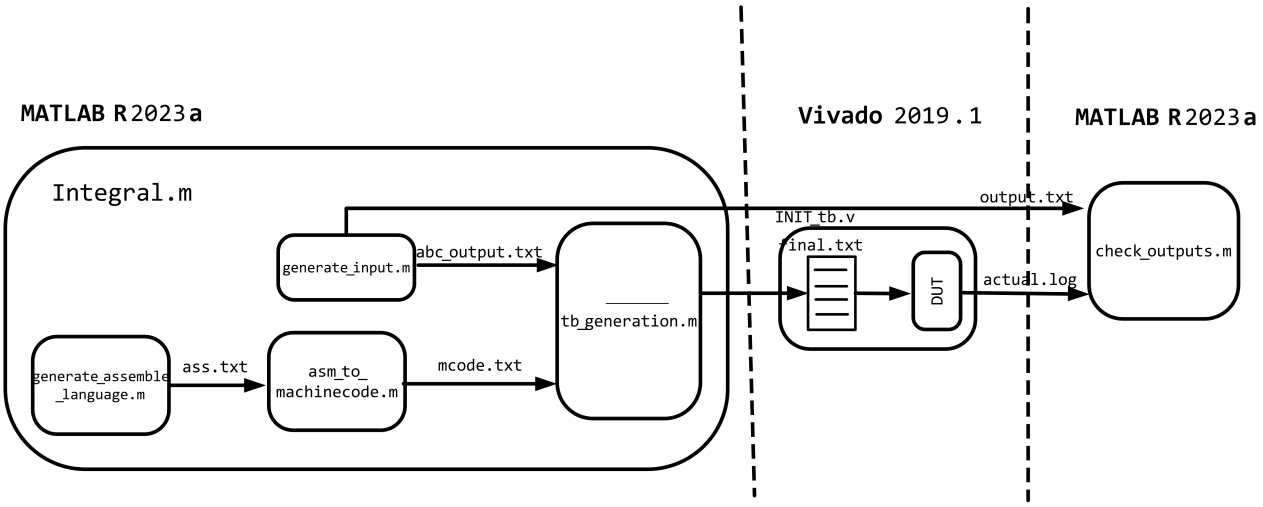

图 6: 系统 setup 与验证流程

从图中可以看到，利用 MATLAB 脚本和 Verilog 语言搭建了联合仿真环境，包括激励生成，仿真运行和结果检查三个部分。其中激励生成和结果检查由 MATLAB 脚本完成，仿真运行由 Verilog 语言实现。

在激励生成模块，有四个子模块。

- Generation_assemble_language.m 用于生成仅与矩阵规模有关的汇编代码
- asm_to_machinecode.m 用于将汇编代码转换为 16 进制机器码
- generate_input.m 随机生成测试激励并通过软件计算得到预期结果
- tb_generation.m 利用机器码和随机生成的测试数据作为 testbench 数据加载阶段的数据来源，生成 testbench 的激励文件。

将设计好的.v 文件在 tb 中实例化得到仿真结果，最后利用 check_outputs.m 检查仿真结果是否正确。具体操作步骤如下：

1. 运行 “Testbench/Matlab_TB_Generation/integral.m” 文件夹下的 integral.m 文件 (应确保文件夹下的其他函数对该文件可见)，生成 testbench 的激励文件 final.txt，其中包括指令和数据，和若干中间文件。
2. 将”Testbench/Testbench/INIT_tb.v” 文件中的激励文件地址替换为实际路径 (不知道为什么好像一定要用绝对路径), 并修改 tb 中的循环轮数参数，利用该 TB 在 Vivado 中进行仿真，得到仿真结果。
3. 将控制窗口中的仿真结果信息复制出来保存为 “Actual.log” 文件，在保证能看见前面生成的中间文件 result.txt 和 Actual.log 文件的前提下，运行 check_outputs.m 脚本，检查仿真结果是否正确。

```
>> check_outputs
已生成 matched.txt，共 8 行实际输出
发现 63 处不匹配，详情见 mismatches.txt
>> check_outputs
已生成 matched.txt，共 64 行实际输出
全部 64 项匹配！
```

图 7: 仿真结果示例

2 仿真验证与性能分析

我们使用 xc7z020clg400-1 芯片进行综合实现，采用的时钟频率为 100MHz。下面的综合实现报告和仿真验证结果

2.1 资源消耗与性能分析

DSP 和存储资源的消耗如表所示：

```
2. Memory
---------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| Block RAM Tile |    0 |     0 |       140 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |       140 |  0.00 |
|   RAMB18       |    0 |     0 |       280 |  0.00 |
+----------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and


3. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |    8 |     0 |       220 |  3.64 |
|   DSP48E1 only |    8 |       |           |       |
+----------------+------+-------+-----------+-------+
```

图 8: 资源消耗分析

更详细的分析可以在 log 文件夹下的 utilization.rpt 文件中查看。

时序和功耗的报告如下：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.057 ns	Worst Hold Slack (WHS):	0.128 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	26470	Total Number of Endpoints:	26470	Total Number of Endpoints:	15358

All user specified timing constraints are met.

图 9: 时序报告

可以看到时序刚刚好不违例，可用的最大时钟周期大约就是 100MHz.

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.311 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**28.6°C**
Thermal Margin:	56.4°C (4.7 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

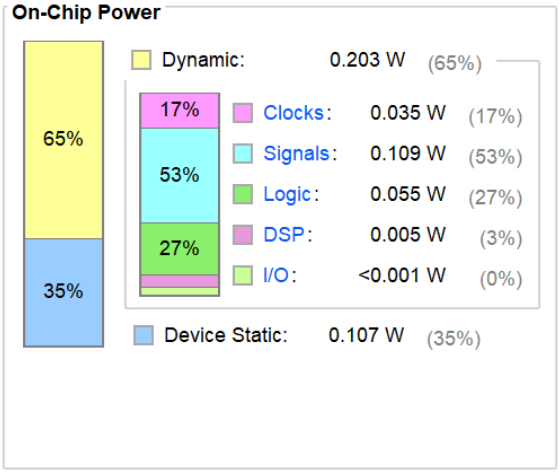

图 10: 功耗报告

下图是宏观的仿真波形：

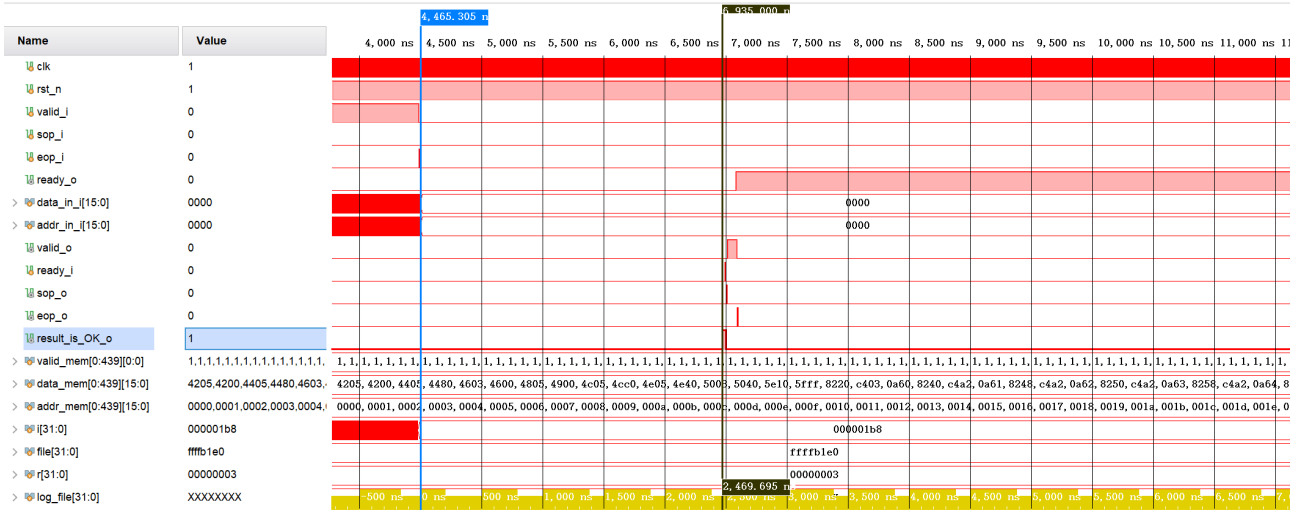

图 11: 吞吐率计算

从波形中不难看出，仿真总共分为三个阶段，中间一个阶段即是计算阶段，对外几乎没有信号输出。给出的波形是矩阵规模为 8*8 的情形，可以看出从开始计算到运算结果全部写回共耗时 2.469μs, 即吞吐率为 0.411GFOPS。

2.2 典型波形与验证结果

2.2.1 验证结果

在给定测试用例下，得到的仿真结果如下图所示：

```
Time=6985000: valid_o=1, data_out_o=0x2ee916a41c36201129f5192627661bf1
Time=6995000: valid_o=1, data_out_o=0x2da81ff9222e2e422f131d8d2a8e2d52
Time=7005000: valid_o=1, data_out_o=0x29241ae81bb21a2d1e4d1dfb1c291b08
Time=7015000: valid_o=1, data_out_o=0x3ca62621283d2f0037f6243037e72994
Time=7025000: valid_o=1, data_out_o=0x1d24106811001f9420ec11141fdb1f11
Time=7035000: valid_o=1, data_out_o=0x317a284e274c28a32a7f264e2a6324dd
Time=7045000: valid_o=1, data_out_o=0x3b3e24db268c2c7a306124bd30b42934
Time=7055000: valid_o=1, data_out_o=0x209019d51c4b214a243f1490201f1e4f
```

图 12: 仿真结果

该结果和软件生成的预期结果完全相同！

2.2.2 仿真波形

下面列举了一些比较有代表性的仿真波形。

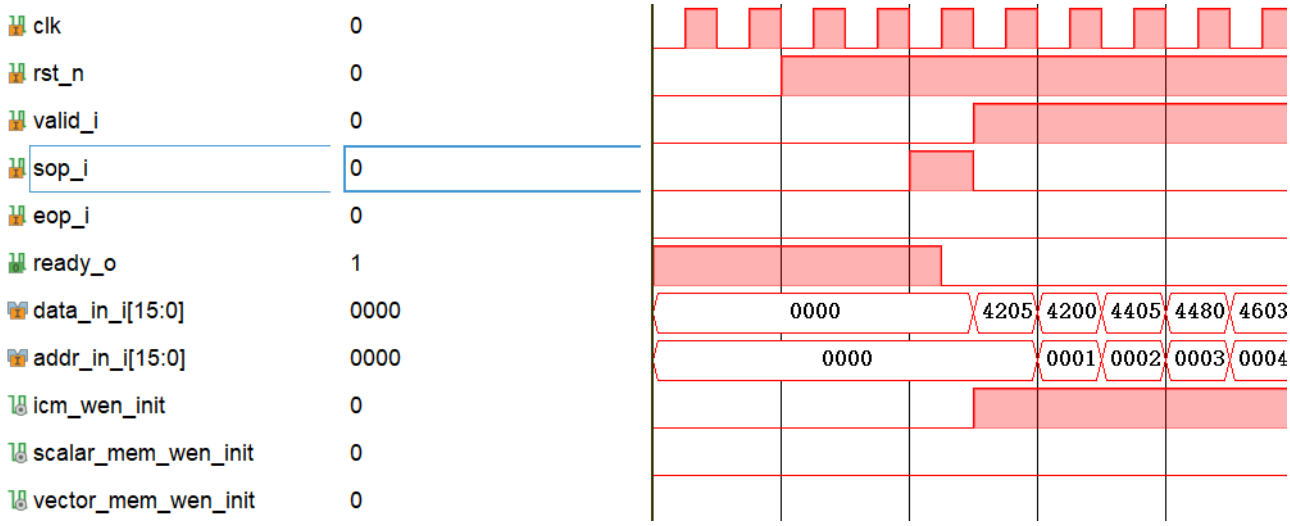

图 13: 输入接口逻辑波形

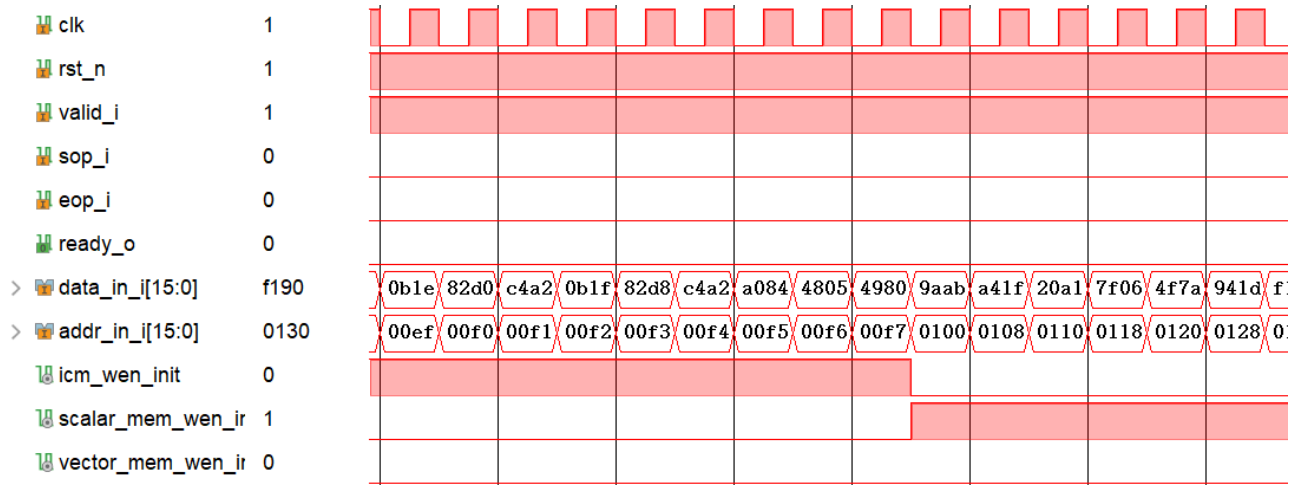

图 14: 地址映射波形

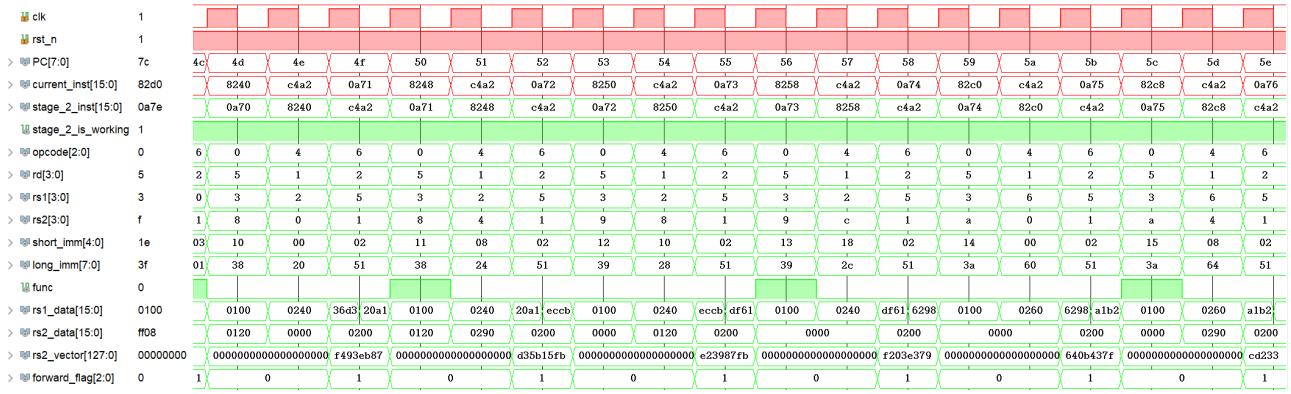

图 15: 四级流水线波形 1

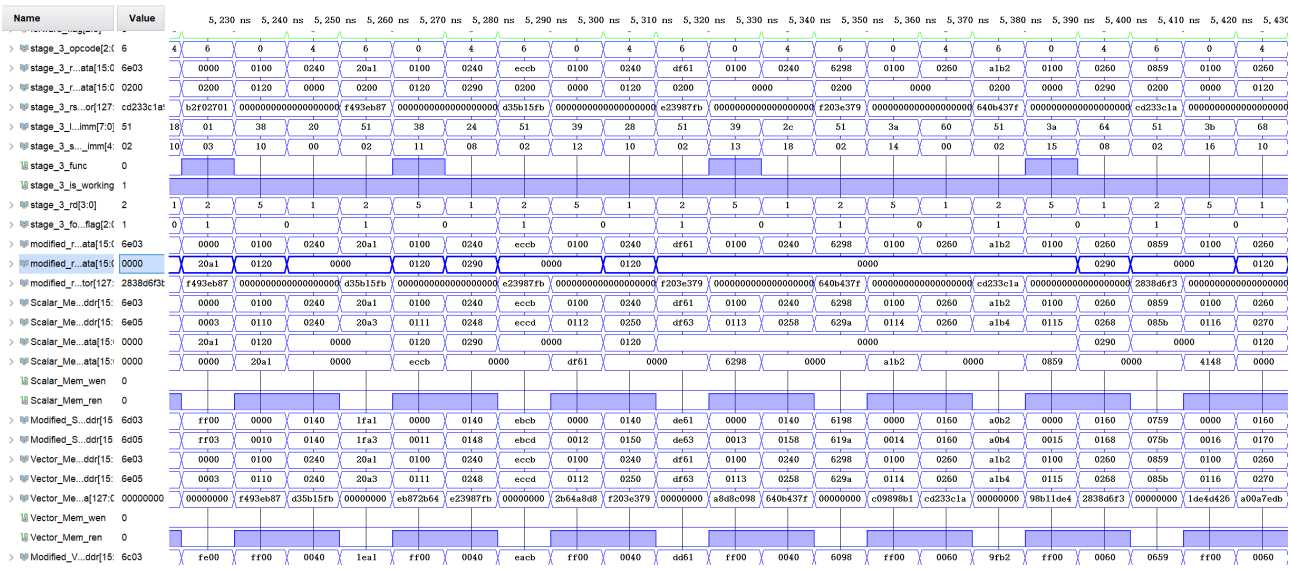

图 16: 四级流水线波形 2

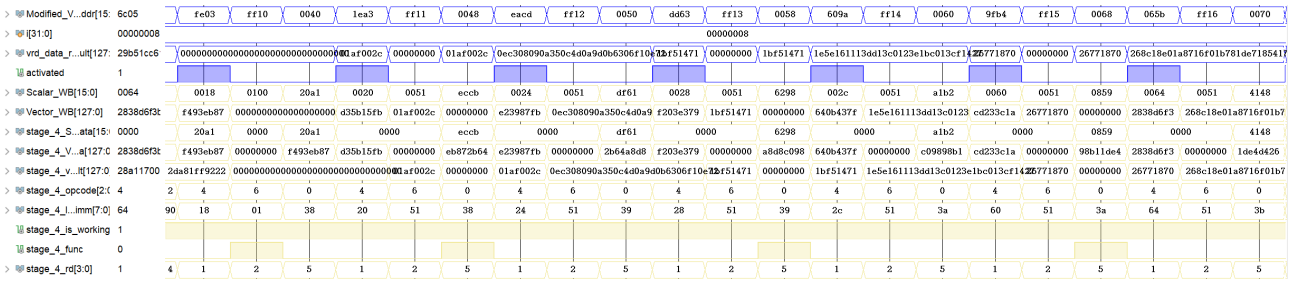

图 17: 四级流水线波形 3

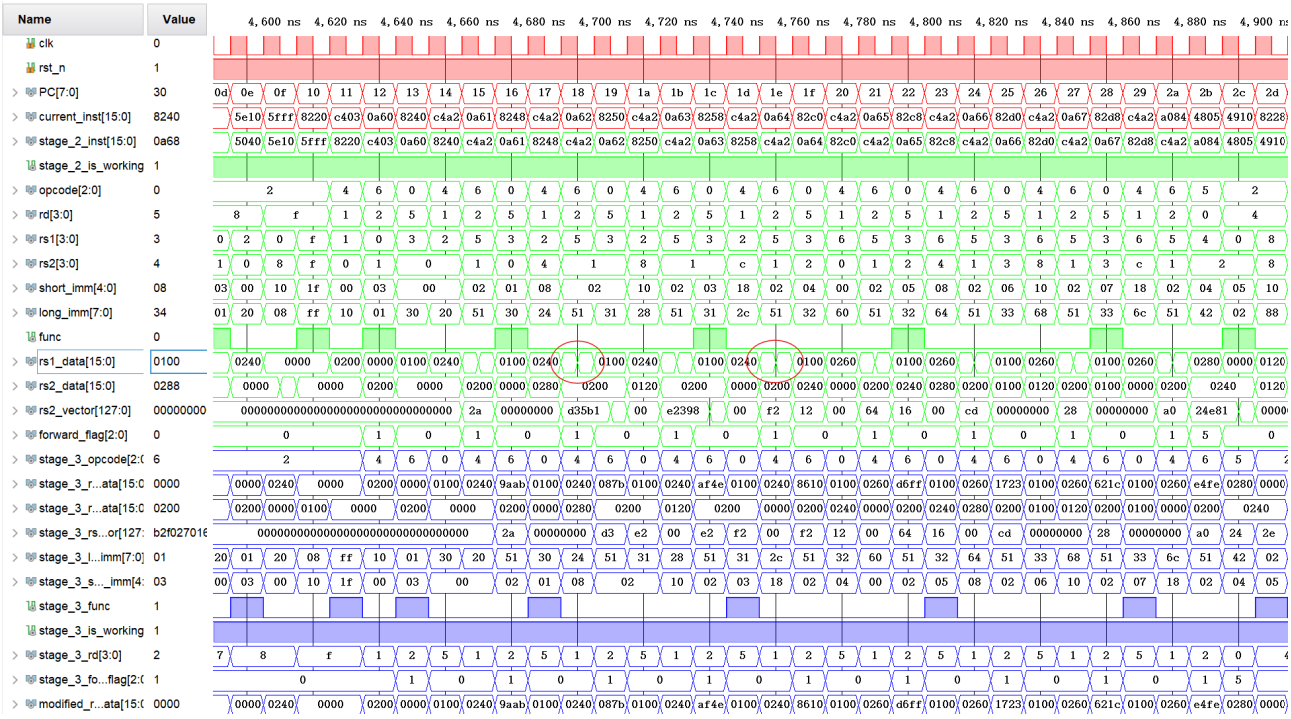

图 18: 下降沿写，上升沿读

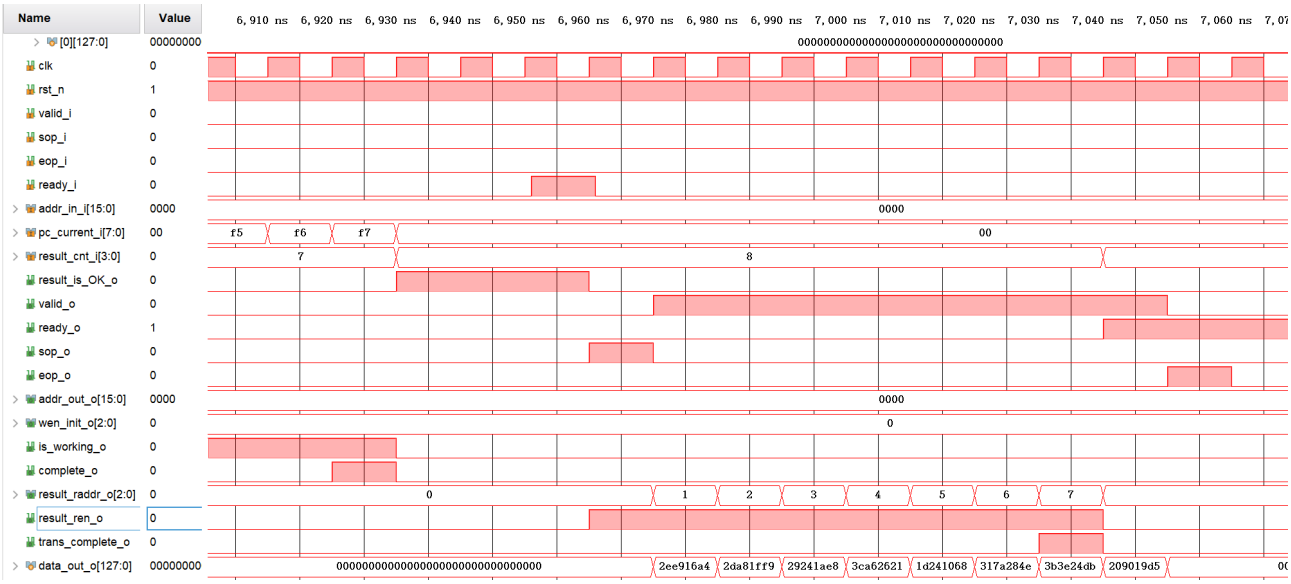

图 19: 输出接口波形

3 优化设计与讨论

从前面的汇编代码不难看出，这样设计的代码存在很多的冗余，比如大量冗余的、用于加载第二个矩阵行的指令。这些矩阵行显然有复用的潜力却因为所在的向量寄存器被反复覆写而需要被反复加载。从这个角度出发，可以充分利用向量寄存器堆，将 16 个向量寄存器堆真正当成缓存来用 (事实上寄存器堆本来就是全相联的缓存)，就可以有效减少指令数量，提高吞吐率。当然由于是软件的优化，不影响硬件结构，硬件的速度和功耗不会发生明显变化 (当然若考虑开关活动率和信号跳转方向的差异可能略有区别)。

具体来说，可以将第二个矩阵的若干个矩阵行存在若干个不同的向量寄存器当中，这样只有计算第一行进行 “暖身” 时需要 vload 加载，其余行均可复用第一次加载的第二个矩阵，减少指令数量，提高吞吐率。部分代码如下：

```
        Vload vr1 r1 0
        VMAC-s vr2 / vr1 $1
        Load r5 r3 0
        Vload vr3 r2 0
        VMAC-s vr2 r5 vr3 $0
        Load r5 r3 1
        Vload vr4 r2 8
        VMAC-s vr2 r5 vr4 $0
        Load r5 r3 2
        Vload vr5 r2 16
        VMAC-s vr2 r5 vr5 $0
        Load r5 r3 3
        Vload vr6 r2 24
        VMAC-s vr2 r5 vr6 $0
        Load r5 r3 4
        Vload vr7 r6 0
        VMAC-s vr2 r5 vr7 $0
        Load r5 r3 5
        Vload vr8 r6 8
        VMAC-s vr2 r5 vr8 $0
        Load r5 r3 6
        Vload vr9 r6 16
        VMAC-s vr2 r5 vr9 $0
        Load r5 r3 7
        Vload vr10 r6 24
        VMAC-s vr2 r5 vr10 $0
        VStore / r4 vr2
        MOV r4 02 $1
        MOV r4 88 $0
        ...
```

仿真得到这个汇编代码的结果是正确的 (结果见 log 文件夹)，且比先前的代码少了将近 20% 的指令，考虑到这还是访存指令，在实际的处理机中耗时会更长，因此这样的修改是非常有好处的。从仿真结果来看，将汇编代码做此优化后，完成整个矩阵乘加运算的用时减少到了 1.911μs，吞吐率提升至 0.569GFOPS，比原先 0.411GFOPS 的吞吐量增加了将近 40%，这个速度提高是非常显著的。

图 20: 优化后的仿真结果

这个优化是一个非常浅显的优化，但总体上效果比较明显。除了这种针对软件的优化外，还可以从硬件的角度出发进行优化，如流水化的计算单元等，可以增大同一时刻在运算单元中进行流动的数据量，进而提高吞吐率。但从 rtl 仿真的层面来看，这种策略的效果不明显，因为存储的带宽基本是够大的。

4 设计总结

本次实验，设计了一款能配置矩阵规模的 SIMD 矩阵运算加速单元，实现了四级流水线的中的数据级并行，并通过设计输入输出接口逻辑和统一存储空间编址在没有板子的情况下模拟了“CPU+ 加速器”系统的仿真。通过 MATLAB 脚本和 Verilog 语言搭建了联合仿真环境，实现了从汇编代码到硬件的综合实现再到仿真的全过程。在这个过程中，我学习了如何使用 Vivado 进行硬件加速器的设计，了解了如何搭建仿真环境和编写测试用例，掌握了如何利用 MATLAB 脚本来实现一个类似于编译器的功能，一定程度上也加深了对算子库和编译器原理的理解。

总的来说，《智能计算芯片导论》这门课作为一个 2 分的课，工作量偏大，在上课的过程中感觉没学到什么东西。但是回味起来，好像把之前计体、嵌入式课上没太搞懂的知识都弄清楚了，讲得很宏观但是细节满满，系统性极强，把观念性思想上的东西讲透彻了，在 2 分课的课时内讲授了 3，4 分课都讲授不出的知识量。这门课对我来讲是收获颇丰的，结合这学期在 FPGA 课上学习的内容以及集创赛备赛过程中的学习，我对软硬件的划分，系统级的设计第一次有了非常清晰的认识，虽说这个学期总体过得浑浑噩噩，但是也不能算是完全没学到东西。超级感谢陈老师精心准备的课程，也感谢助教老师们的耐心指导！