OpenTensorCore 架构介绍

X. Shen W. Chen

1 基础知识概述

2 项目目标

3 项目框架介绍

4 基本运算模块

5 未来改进方向

设计TensorCore的意义

a) 什么是TensorCore?

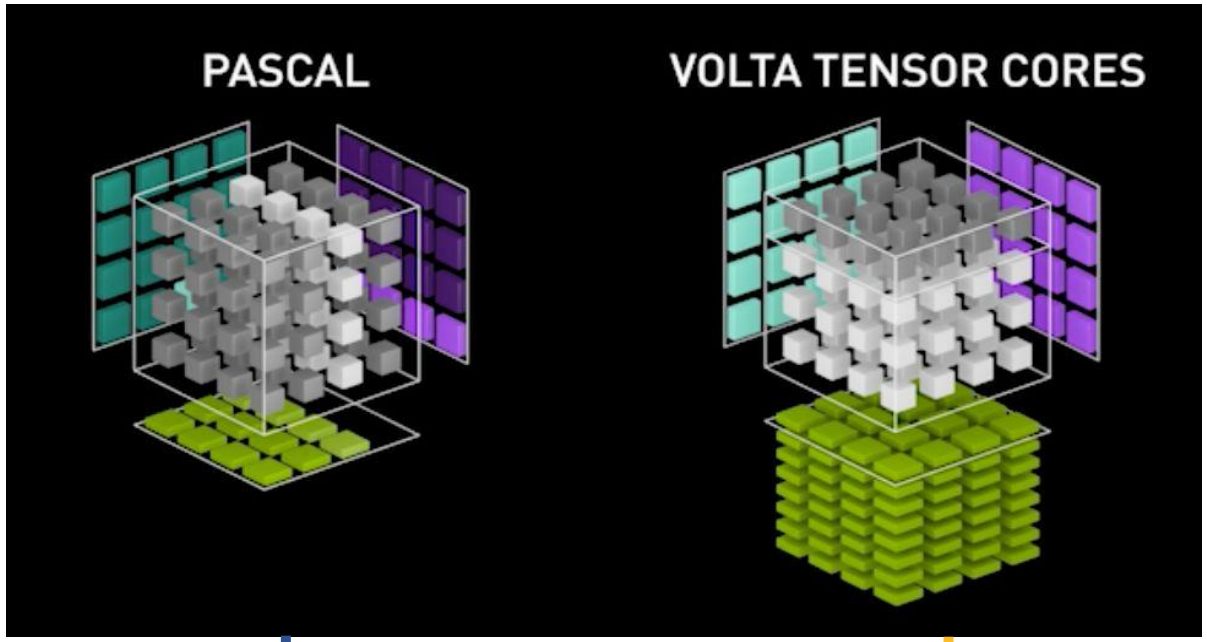
- ✓ 可以实现**混合精度计算**并**加速矩阵运算**的专用硬件单元
- ✓ 尤其擅长处理半精度（FP16）和全精度（FP32）的**矩阵乘法和累加操作**

b) 核心目标?

- ✓ 通过**并行化和流水线技术**加速矩阵乘法和累加操作（GEMM)

c) 核心特性

- 混合精度支持
- 高吞吐量：支持数据搬运与计算并行
- 灵活的数据类型
- 内存高效性



无TensorCore：每时钟周期以元素乘行的方式，4次相乘得一列数据

有Tensor Core：直接计算矩阵A与B的乘积，一次性输出完整矩阵

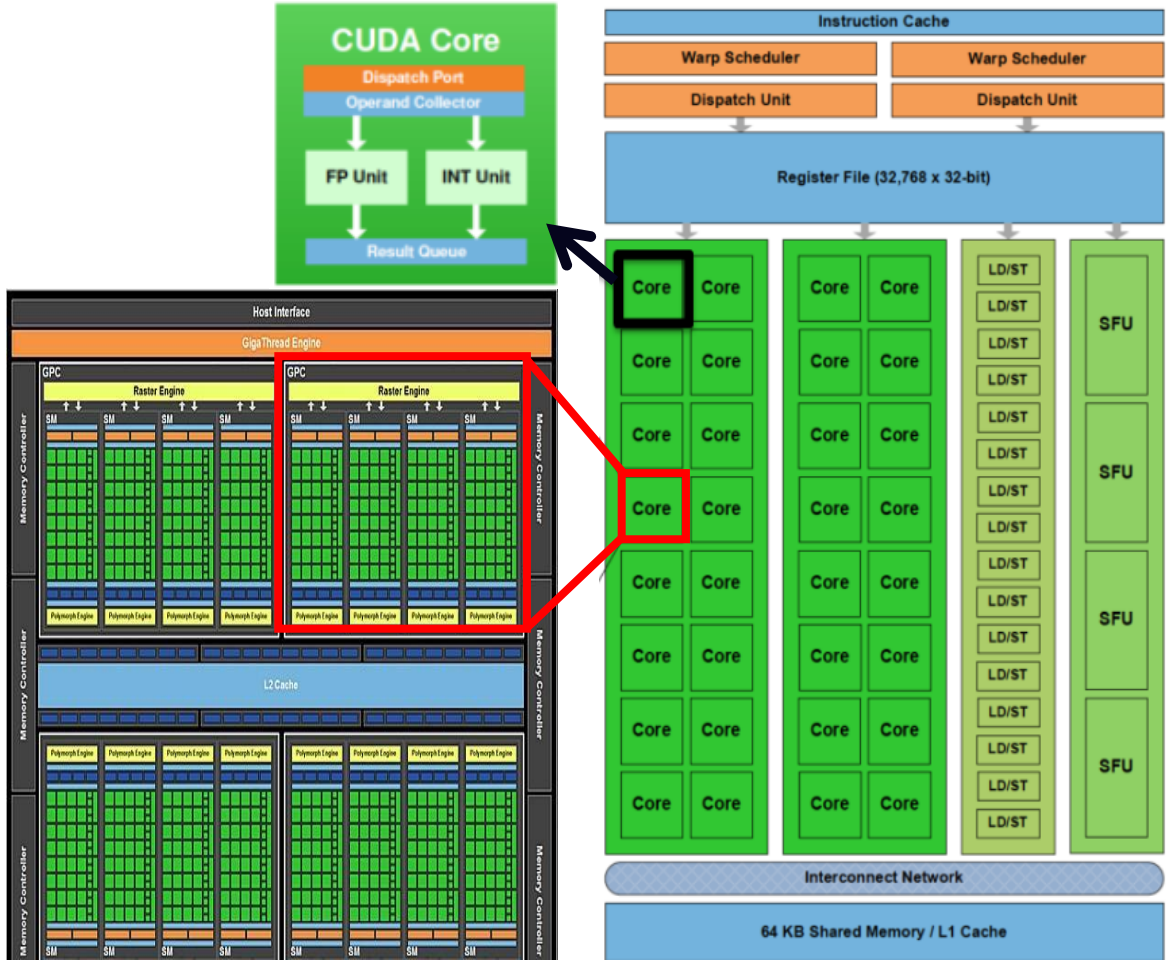
英伟达GPU架构的演进

在英伟达的通用 GPU 架构中，主要存在三种核心类型：**CUDA Core、Tensor Core 以及 RT Core**

- ✓ 2006 年，在 Tesla 架构中首次提出 CUDA Core
- ✓ 2017 年，在 Volta 架构中首次提出 Tensor Core
- ✓ 2018 年，在 Turing 架构中首次提出 RT Core

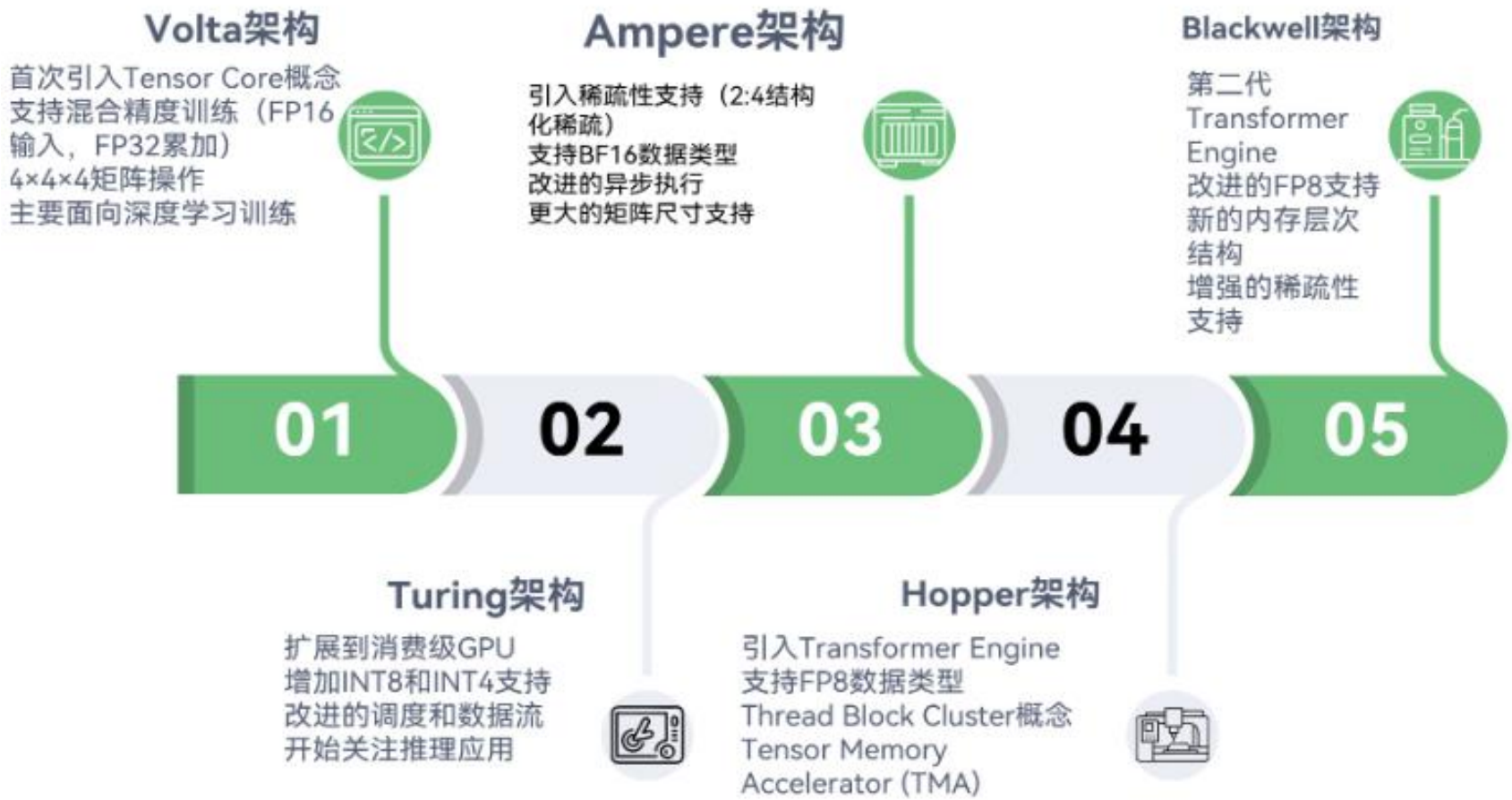
处理核心：SP(Stream Processor) → CUDA Core

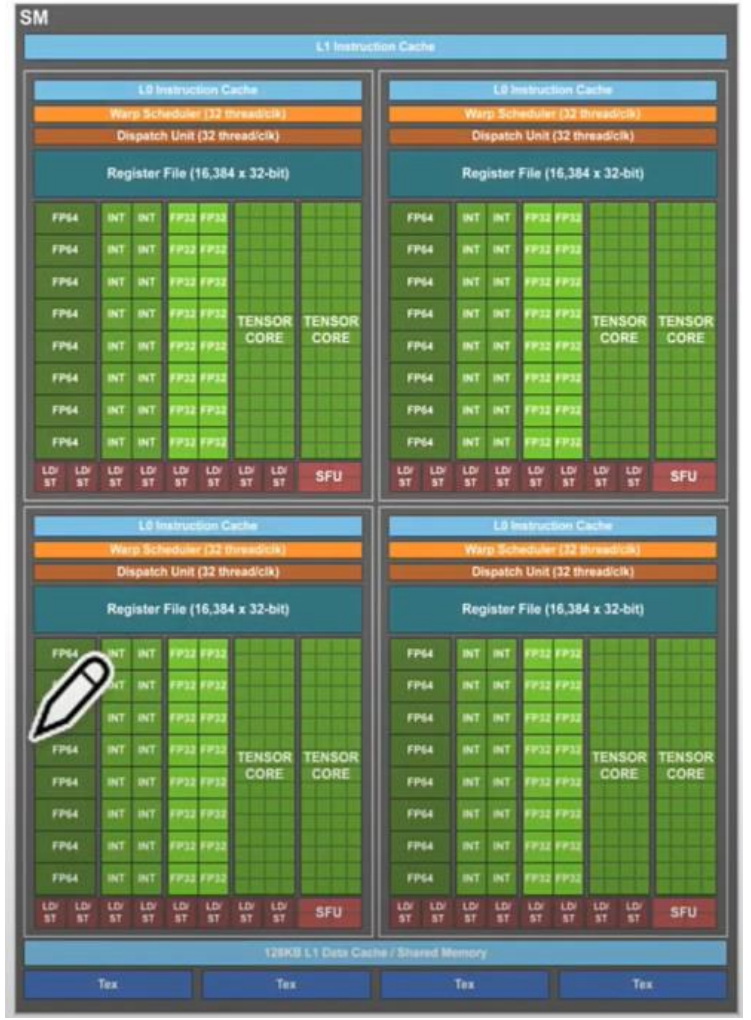
架构名称	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
中文名字	费米	开普勒	麦克斯韦	帕斯卡	伏特	图灵	安培	赫柏
发布时间	2010	2012	2014	2016	2017	2018	2020	2022
核心参数	16个SM，每个SM包含32个CUDA Cores，一共512 CUDA Cores	15个SMX，每个SMX包括192个FP32+64个FP64 CUDA Cores	16个SM，每个SM包括4个处理块，每个处理块包括32个CUDA Cores +8个LD/ST Unit + 8 SFU	GP100有60个SM，每个SM包括64个CUDA Cores，32个DP Cores	80个SM，每个SM包括32个FP64 +64 Int32+64 FP32+8个Tensor Cores	102核心92个SM，SM重新设计，每个SM包含64个Int32+64个FP32+8个Tensor Cores	108个SM，每个SM包含64个FP32+64个INT32+32个FP64+4个Tensor Cores	132个SM，每个SM包含128个FP32+64个INT32+64个FP64+4个Tensor Cores
特点&优势	首个完整GPU计算架构，支持与共享存储结合的Cache层次GPU架构，支持ECC GPU架构	游戏性能大幅提升，首次支持GPU Direct技术	每组SM单元从192个减少到每组128个，每个SMM单元拥有更多逻辑控制电路	NVLink第一代，双向互联带宽160 GB/s，P100拥有56个SM HBM	NVLink2.0，Tensor Cores第一代，支持AI运算	Tensor Core2.0，RT Core第一代	Tensor Core3.0，RT Core2.0，NVLink3.0，结构稀疏性矩阵MIG1.0	Tensor Core4.0，NVlink4.0，结构稀疏性矩阵MIG2.0



Fermi架构

TensorCore的演进

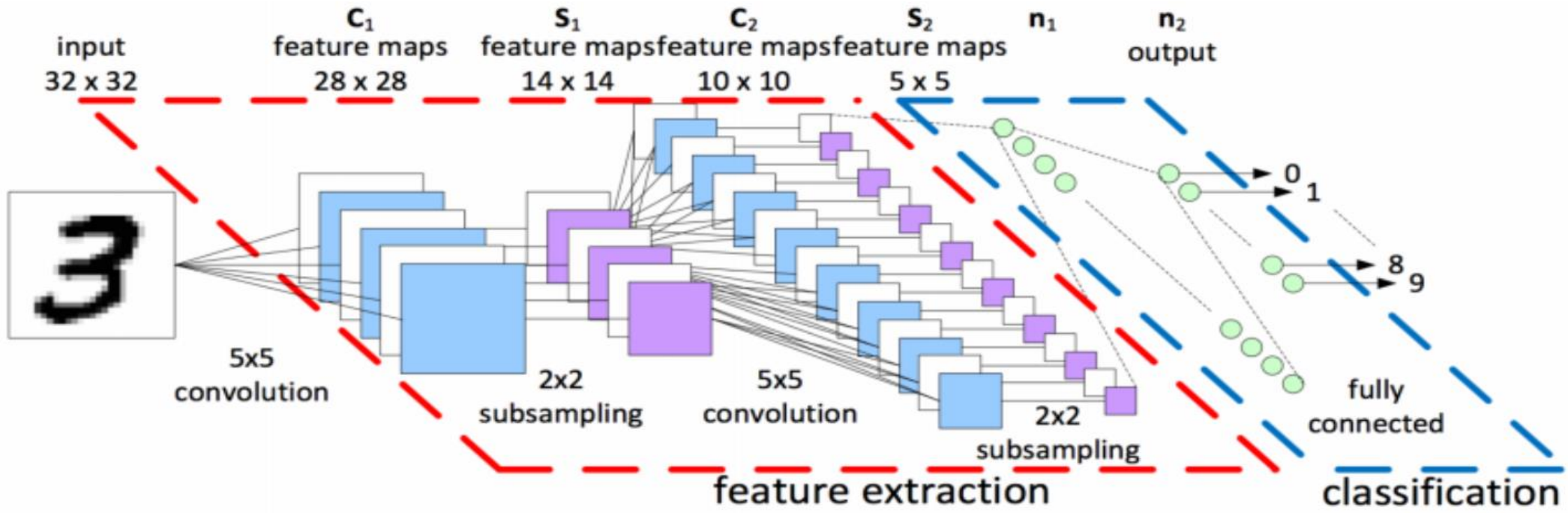




具体演进及各代TensorCore结构详见 https://zhuanlan.zhihu.com/p/695336039

卷积计算

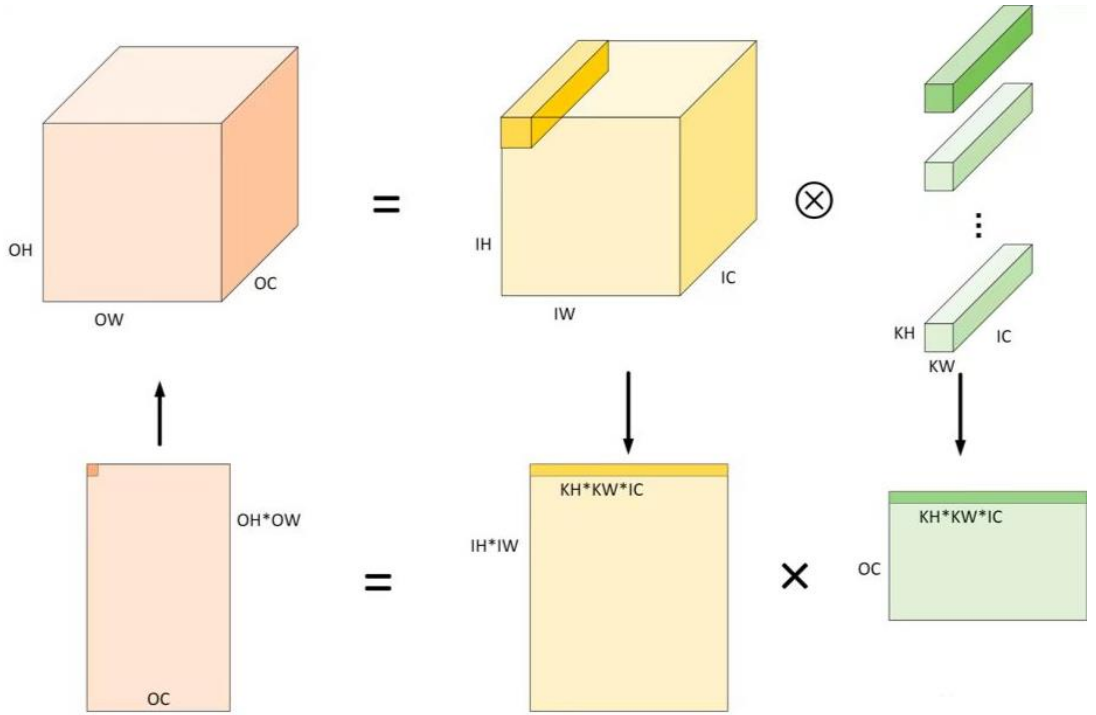
- 功能：可用于处理图像、语音、文本等数据，通过**提取并学习数据中的特征**，从而实现图像识别、分类、分割等任务
- 卷积运算通常与激活函数（如 ReLU）、池化层等结合使用，构成卷积神经网络（CNN）



Im2Col算法

数据重排（**Im2Col操作**）：将卷积运算转换为矩阵乘法

- 好处：1、可利用GEMM加速卷积计算；
  2、可以减少重复的内存访问
- 步骤：1、使用 Im2Col 将输入矩阵展开一个大矩阵，矩阵每一列表示卷积核需要的一个输入数据
  2、再进行 Matmul 运算，得到的数据就是最终卷积计算的结果
- 结果：原本的卷积运算转化为两个矩阵的乘法操作

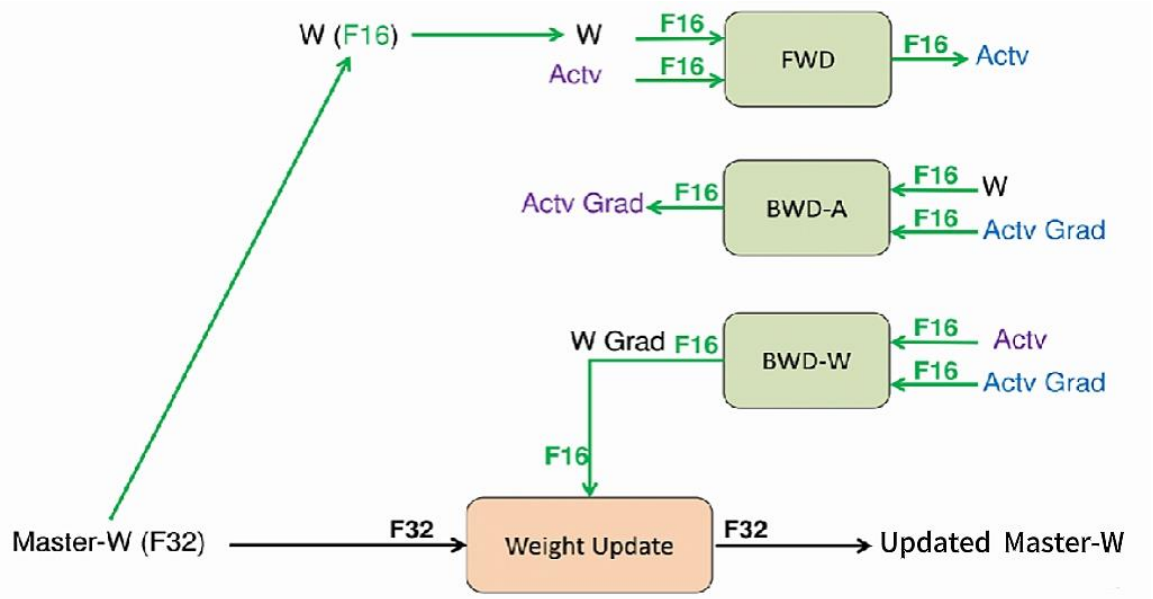


混合精度训练

通过在模型训练过程中灵活地使用不同的数值精度来达到加速训练和减少内存消耗的目的

1、计算的精度分配：在模型的传播过程中，使用较低的精度（如 FP16）进行计算

2、参数更新的精度保持：更新模型参数时，使用较高的精度（如 FP32）来保持训练的稳定性和模型的最终性能

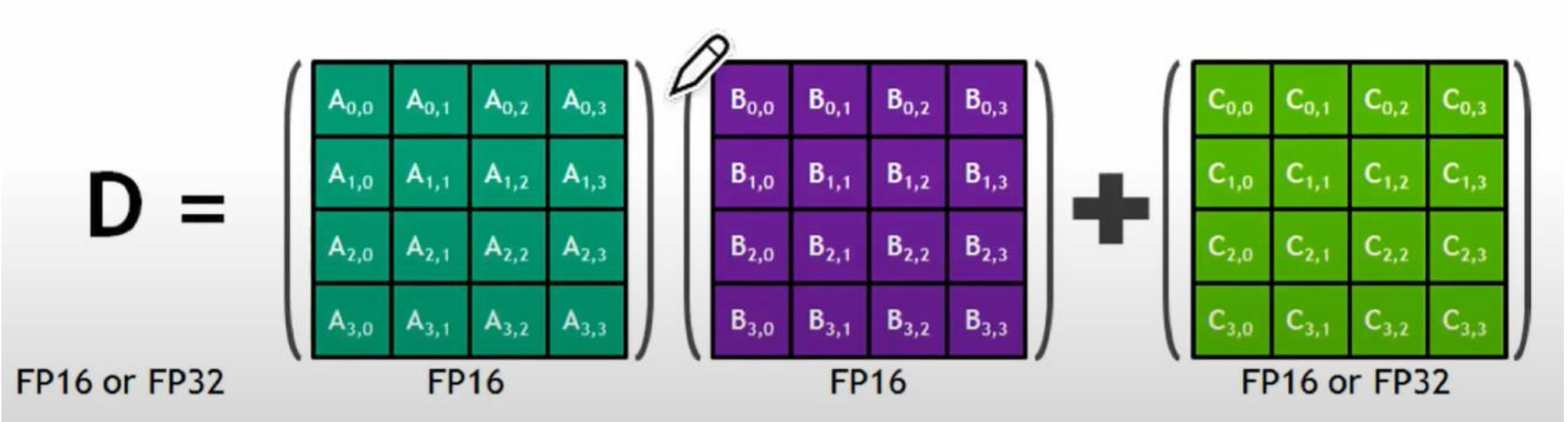


Tensor Core 就是专门设计来加速 FP16 计算的，同时保持 FP32 的累加精度。在底层硬件算子层面，使用半精度(FP16)作为输入和输出，使用全精度(FP32)进行中间结果计算从而不损失过多精度

TensorCore基本原理

a) 采用融合乘法加法（FMA），每个 Tensor Core 每周期能执行 4x4x4 GEMM，64 个浮点乘法累加（FMA）运算

b) 矩阵乘法输入 A 和 B 是 FP16 矩阵，而累加矩阵 C 和 D 可以是 FP16 或 FP32 矩阵



设计TensorCore的目标

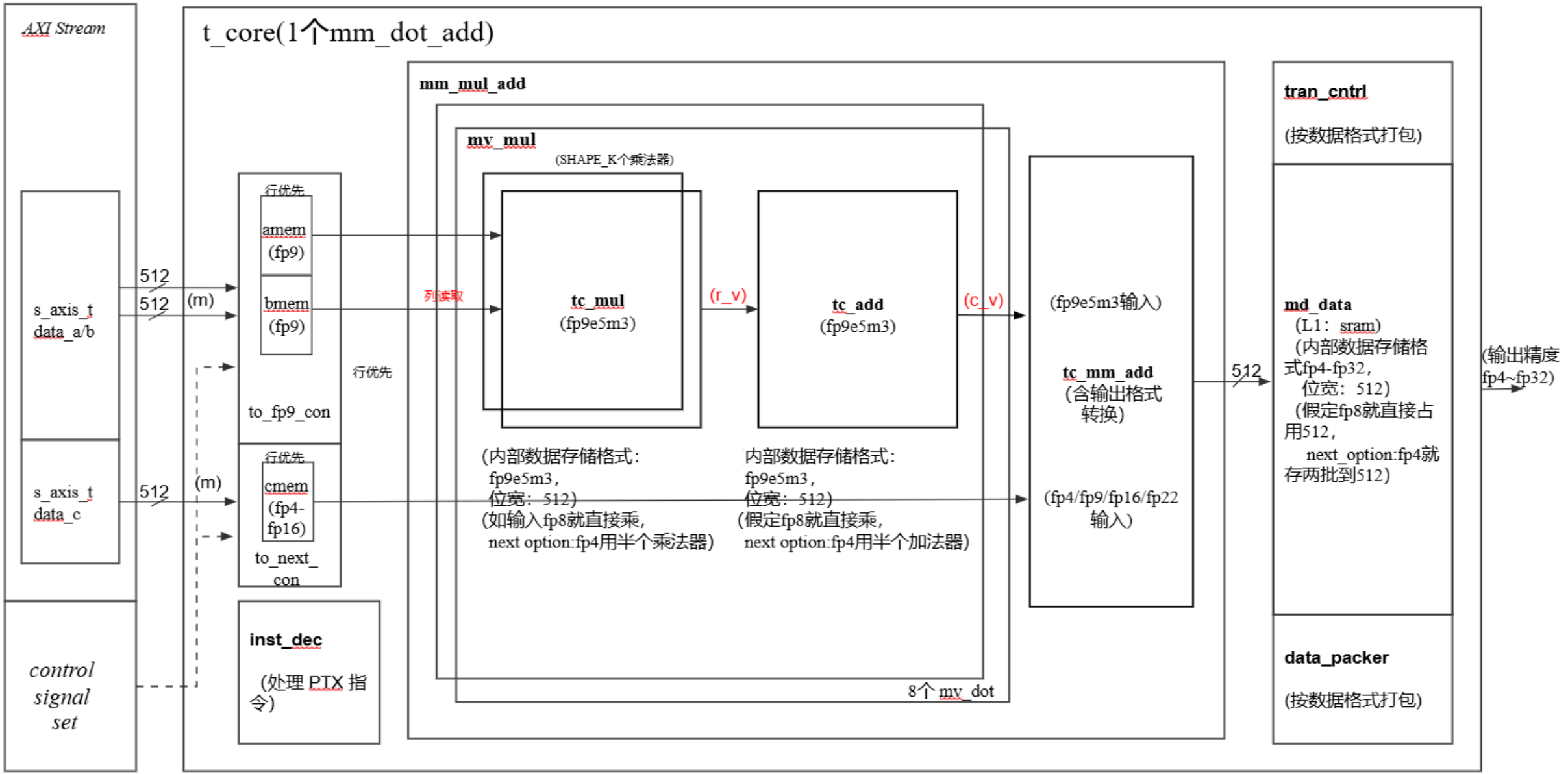
a) 兼容现有gpgpu-sim里的函数接口模式，方便编译器开发者与开源社区为该TensorCore适配CUDA编译器。

b) 在兼容gpgpu-sim的前提下支持FP8和FP4，方便性能提升。

c) 未来能整合入vortex。

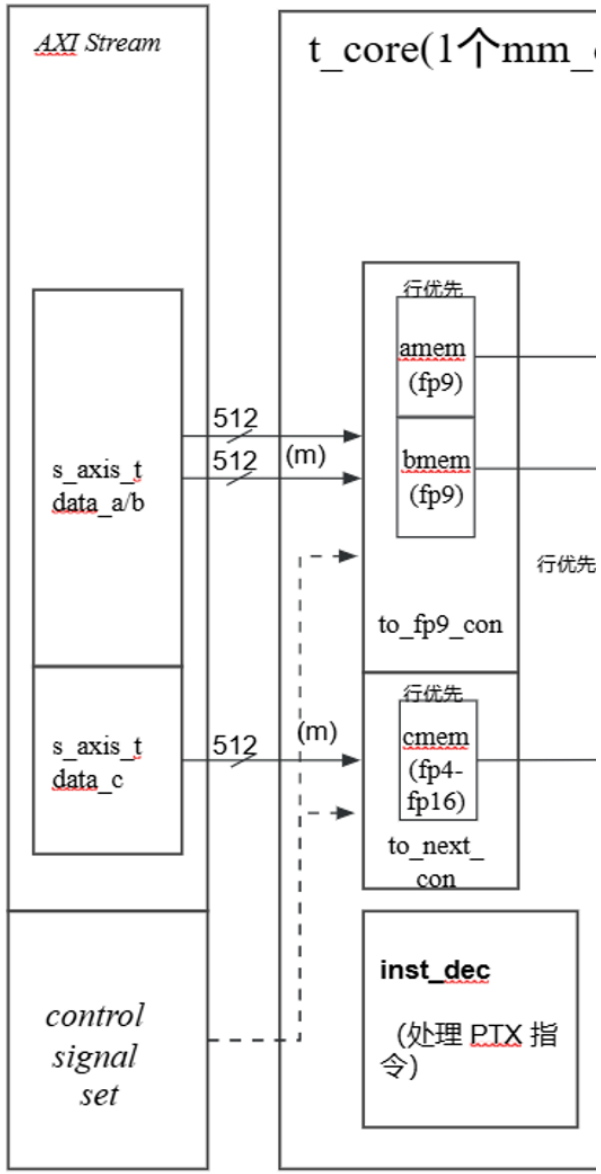
d) MAC或TensorCore本体，标准化，方便嵌入各种设计。

TensorCore 整体架构



TensorCore 整体架构

输入控制接口



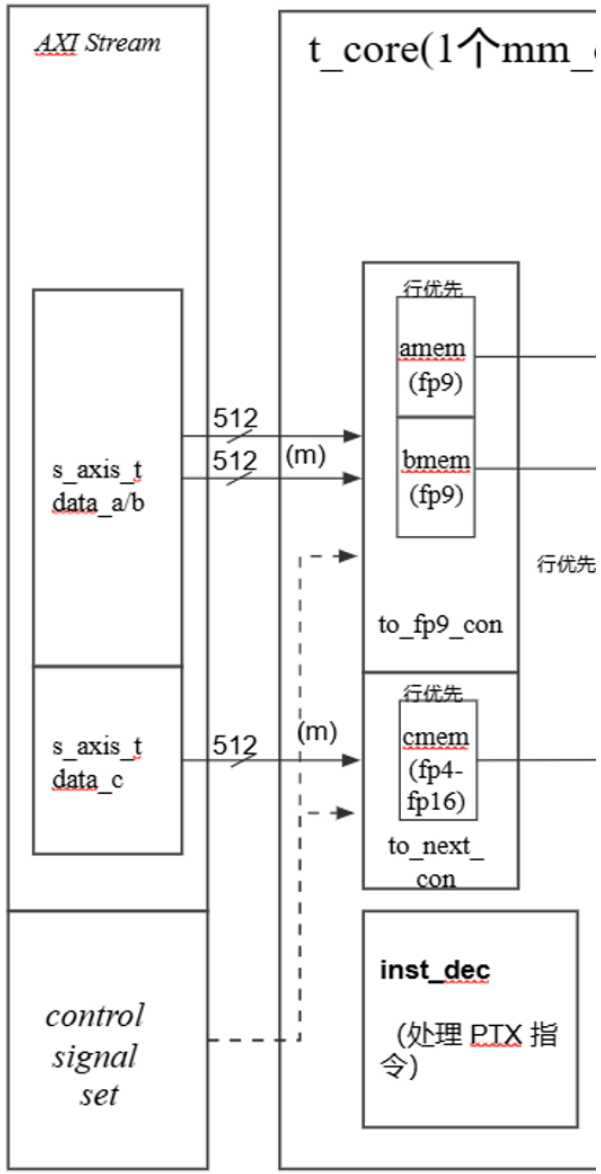
输入控制接口模块

- AXI Stream (AXIS) 接口：负责与外部系统进行高速数据交互
  - ✓ 极简架构
    - ➢ 移除地址通道（AW/AR）与响应信号（BRESP/RRESP），仅保留数据流核心。
    - ➢ 半双工设计，同一时刻仅支持单向传输（读或写）
  - ✓ 流式数据处理优势
    - ➢ 支持无限连续传输（无突发长度限制），适配实时数据流需求

- s_axis_t data_a/b：
  - ✓ 用于输入主矩阵数据（矩阵 A 和矩阵 B）
- s_axis_t data_c
  - ✓ 用于输入累加/输出初值

- 控制信号集（control signal set）
  - ✓ 负责指令解析和运行时配置
  - ✓ 完成数据的装载、计算和输出

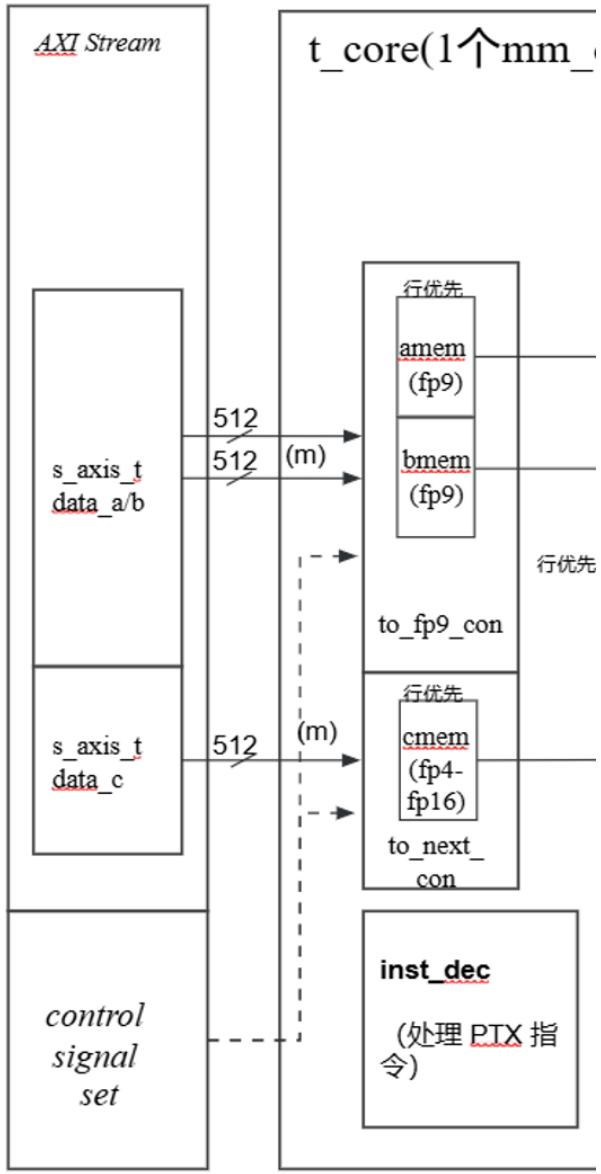
输入控制接口



输入控制接口模块

- inst_dec模块
  - 解析上层发来的 PTX 指令。
  - PTX 是 CUDA 编译器（nvcc） 生成的 GPU 汇编语言，用于在不同硬件架构之间提供一定的可移植性。
- to_fp9_con模块
  - 接收ab数据，送入计算路径
  - 根据数据精度选择不同格式转换通路，最终转换为fp9（e5m3）格式存到sram模块
  - 若总线数据精度为fp4，则需要并串转换存2次，若数据精度为fp16，则计算2个周期存1次
- to_next_con模块
  - 接收c数据，起结果搬运的作用，根据tc_mm_add的输入精度选择不同的格式将输入转换为fp4-fp22

存储分层设计



输入控制接口模块

- 计算核心内部通过 多级片上存储（SRAM） 来缓冲和复用数据，减少对外部 DRAM 的访问。这里主要分为三类：

存储块	数据类型	位宽	功能定位
amem	矩阵 A	fp9	行优先存储 A 矩阵数据
bmem	矩阵 B	fp9	行优先存储 B 矩阵数据
cmem	矩阵 C / 中间结果	fp4–fp16	存储计算结果或累加值

- 设计要点

✓ Tile 结构

➢ 每个寄存器 Tile 拆分为 32 个子块，每块存放固定长度数据（约 16 B）。数据在各子块间平均分配以实现负载均衡

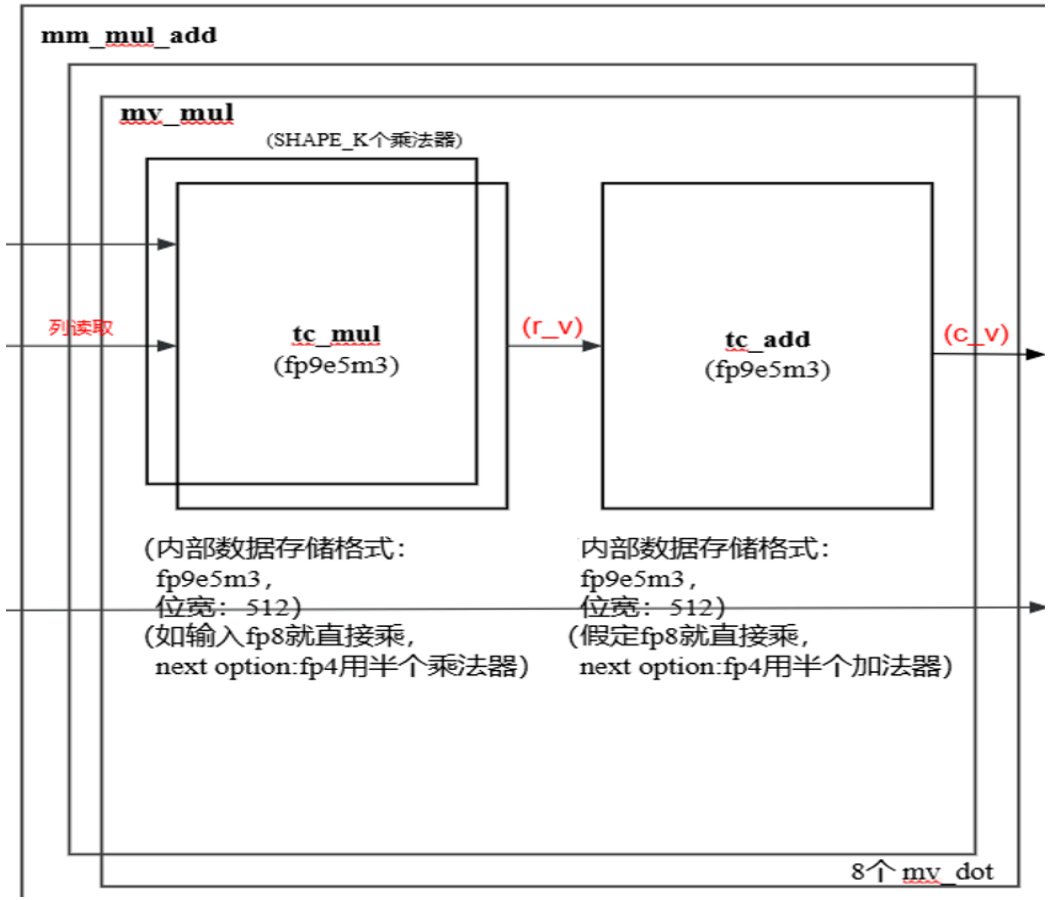
✓ 地址映射与补齐

➢ 当数据不足 32 块时，通过地址映射逻辑执行自动补齐。使用 texture 拷贝机制完成，不增加访存开销

✓ 访问特性

➢ 采用连续 bit 单元存储，确保最小存储粒度的一致性。多路并行访问接口与 MAC 阵列深度绑定，实现高吞吐加载

矩阵乘加模块



矩阵乘加模块

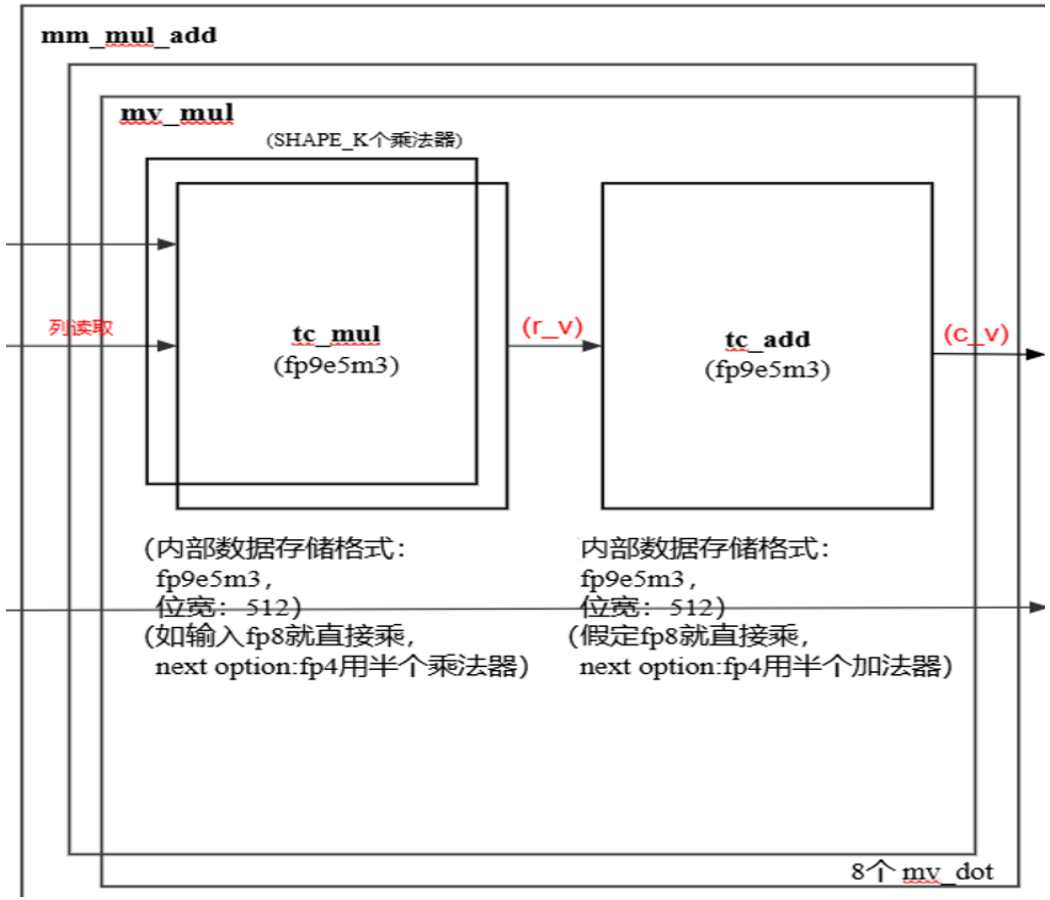
- mm_mul_add模块
  - ✓ 该模块是一个完整的矩阵乘加运算，是gemm分解为gemv算法的实现

$$C = A \cdot B + C_0$$

  - ✓ 支持混合精度计算（如 FP16/BF16 /FP8输入、FP32 累加）
  - ✓ 由多个并行的Mac单元组成，每个单元负责一个小矩阵块的计算。
  - ✓ 支持 Warp 级别的并行计算，每个Warp的线程配置1-8个Mac阵列。
  - ✓ 每个Mac运算单元每个周期处理1个32位存储中的A，B输入累加

- 底部标注了 “8 个 mv_dot”
  - ✓ 表明硬件内部一次性支持 8 路矩阵-向量乘法并行

矩阵乘加模块



矩阵乘加模块

- mv_mul模块

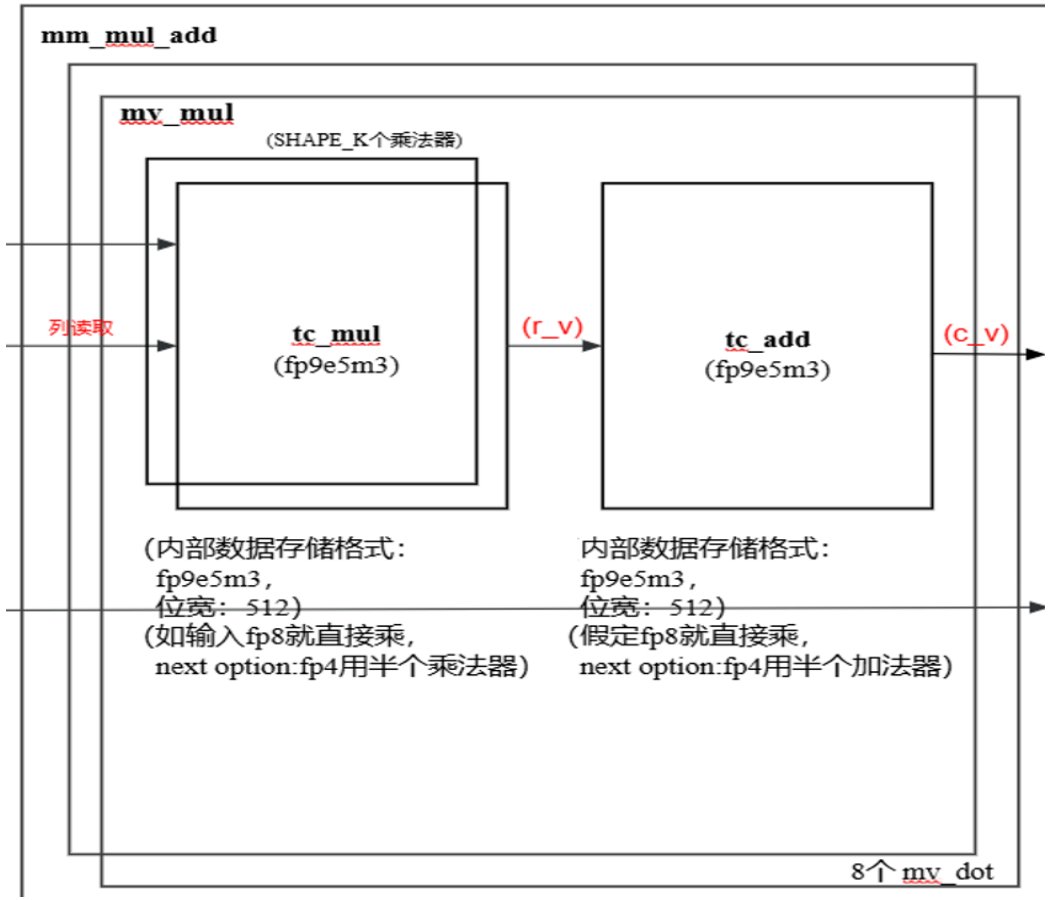
该实现了gemv的功能，大致流程如下：

a) 例化tc_mul模块，用于A的一行和B的一列相乘

$$C_{m\times n} = A_{m\times k}B_{k\times n}$$

b) 例化tc_add模块，实现输入行向量组的求和，此时输出为shape_k*1的列向量

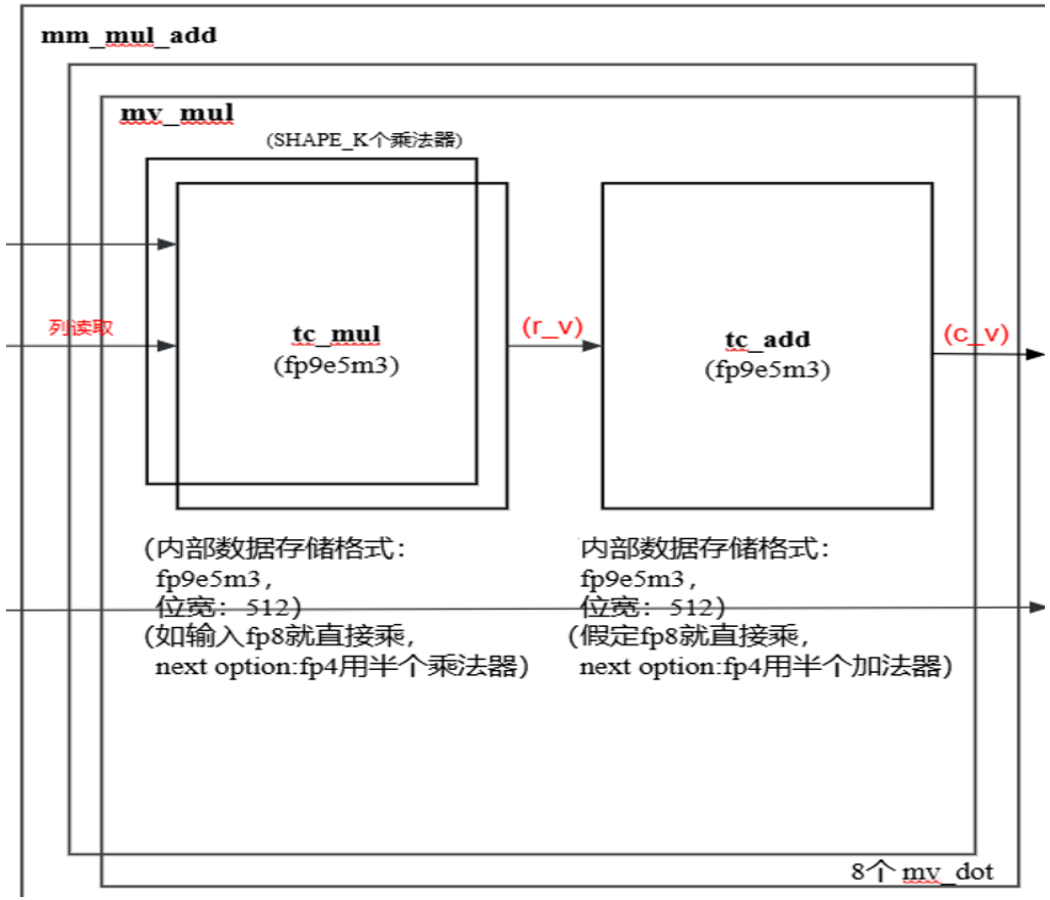
矩阵乘加模块



矩阵乘加模块

- tc_mul
  - ✓ 使用格式：fp9(e5m3)
  - ✓ 位宽：512 bit（与外部 AXI Stream 保持一致）
  - ✓ Shape_k是tc_mul需要的乘法器个数，即A的列数
  - ✓ 支持 fp8 输入直通
    - ➢ 说明乘法阵列可以自动适配更低精度输入，体现出“混合精度”能力
  - ✓ 未来fp4 用半个乘法器
    - ➢ 表明可通过精度降低来进一步复用硬件资源（半精度共享一个乘法器）

矩阵乘加模块



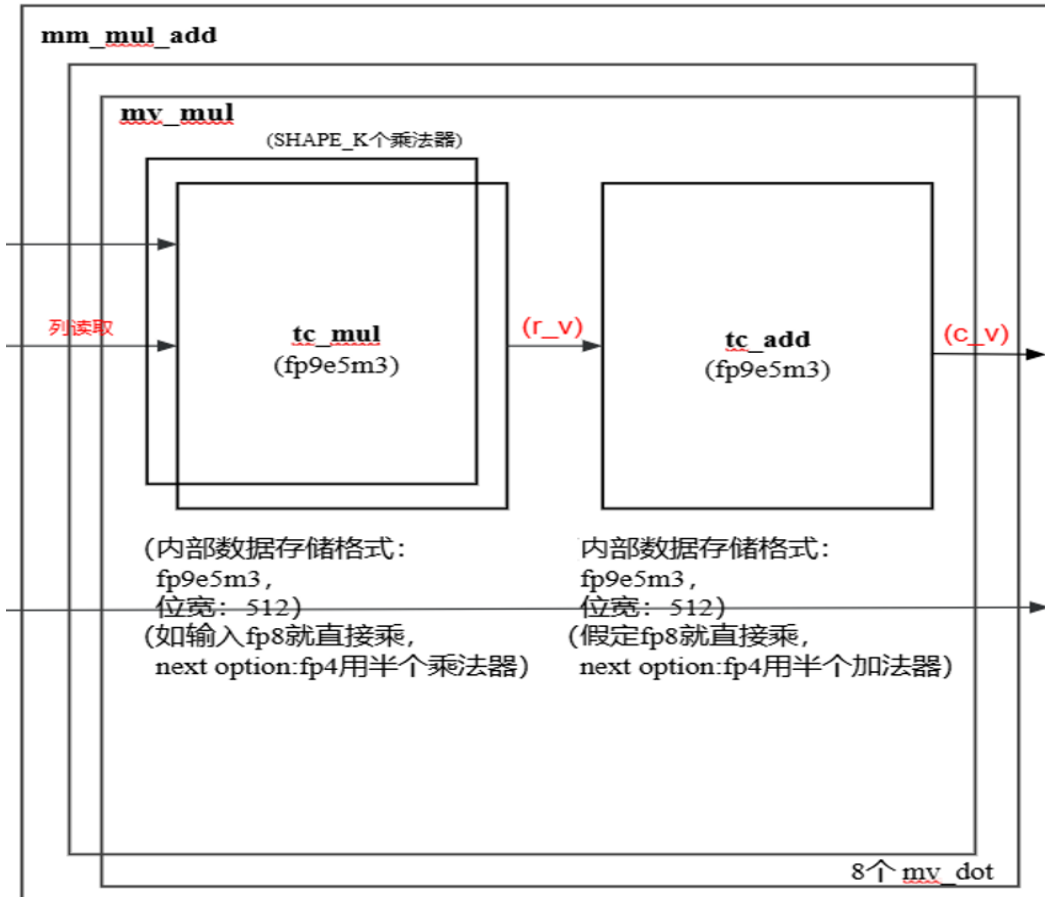
矩阵乘加模块

- tc_mul

乘法流水线分3级实现，用反压信号控制流水线寄存器更新

- 预处理阶段（fmul_s1）和尾数相乘（naivemultiplier）
- 第一级输出整合阶段（fmul_s2）
- 舍入与规格化阶段（fmul_s3）
- 扩位至fp22的中间结果r_v

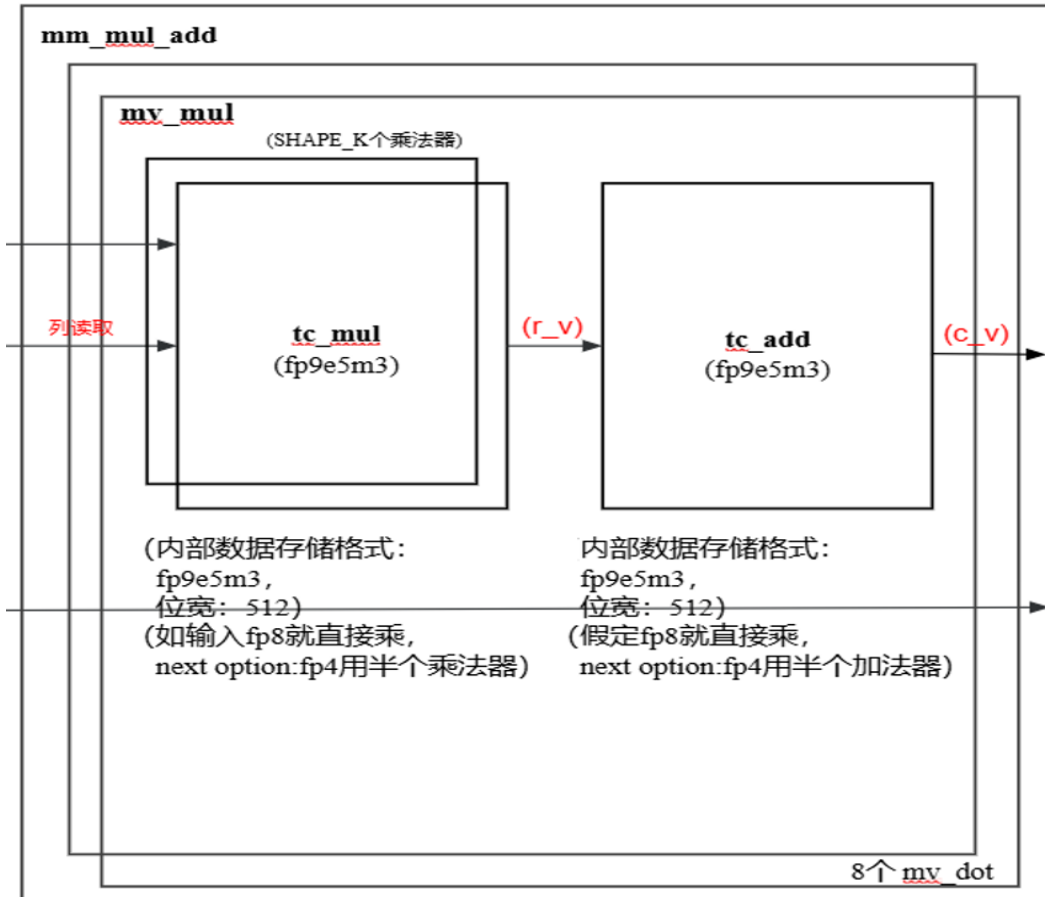
矩阵乘加模块



矩阵乘加模块

- tc_add
  - ✓ 输入：tc_mul输出的一个行向量组
  - ✓ 操作：对这个行向量组的每一行进行累加
  - ✓ 输出：列向量 c_v

矩阵乘加模块

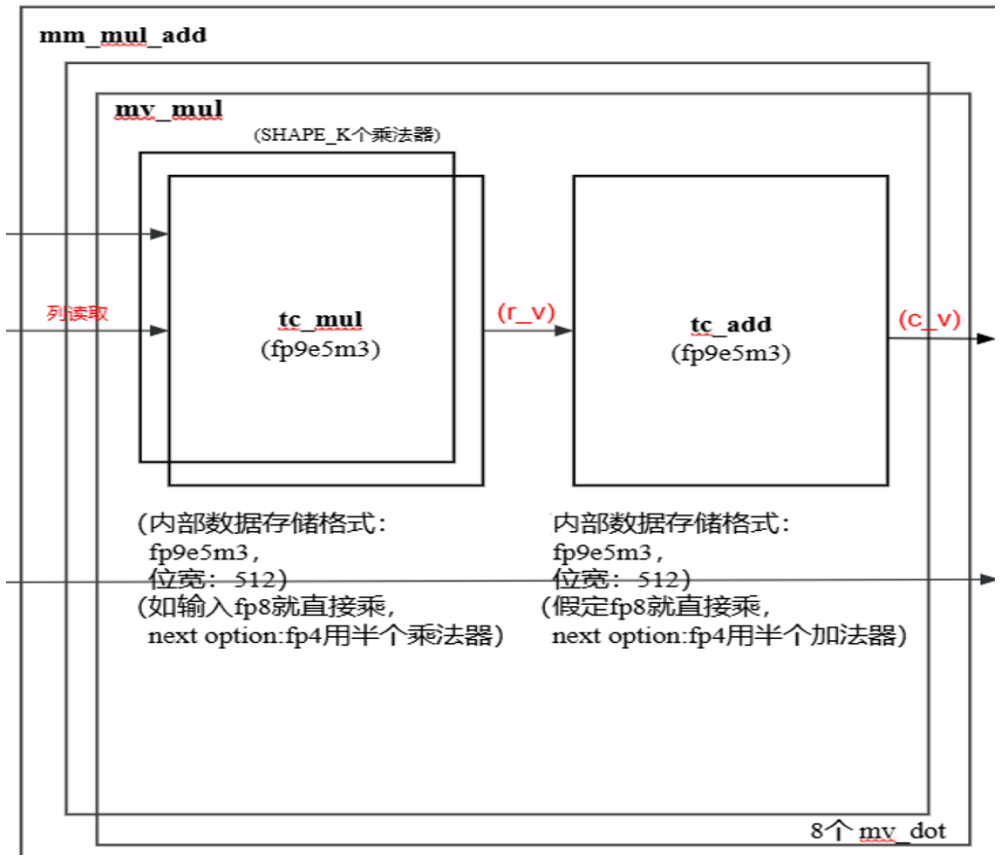


矩阵乘加模块

- 加法流水线分2级实现
  - fadd_s1对两个浮点数分类，选择计算路径相加
  - fadd_s2处理舍入，溢出和规格化

运算过程数据流向

输入矩阵列读取 → tc_mul (乘法) → r_v 中间结果 → tc_add (累加) → c_v 输出结果



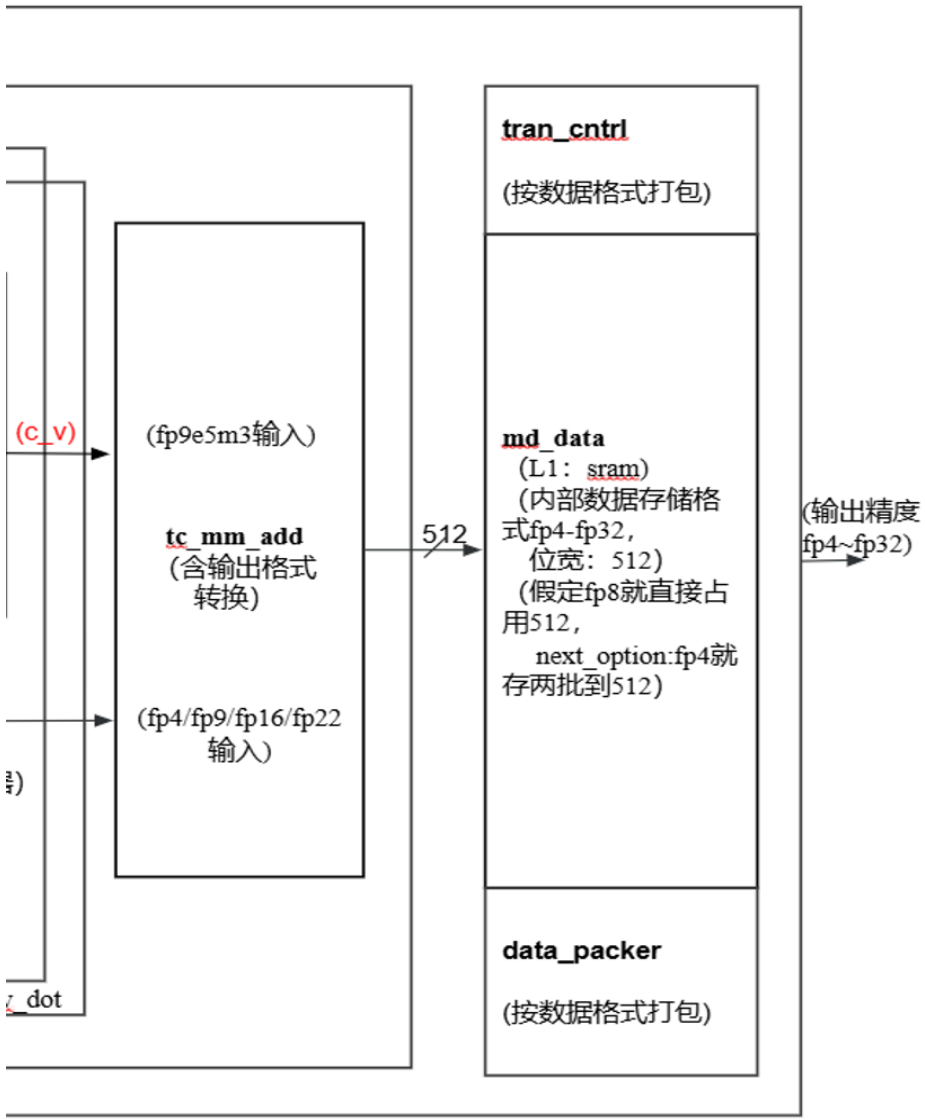
矩阵乘加模块

- 对标 GEMM 的数学式:

$$C_{m\times n} = \sum_{k=1}^{K} A_{m\times k} \times B_{k\times n}$$

- “列读取” 对应矩阵 B 的按列访问
- mv_mul 拆分成多个矩阵-向量乘法（每列对应一个向量）
- tc_mul 实现乘法
- tc_add 完成累加

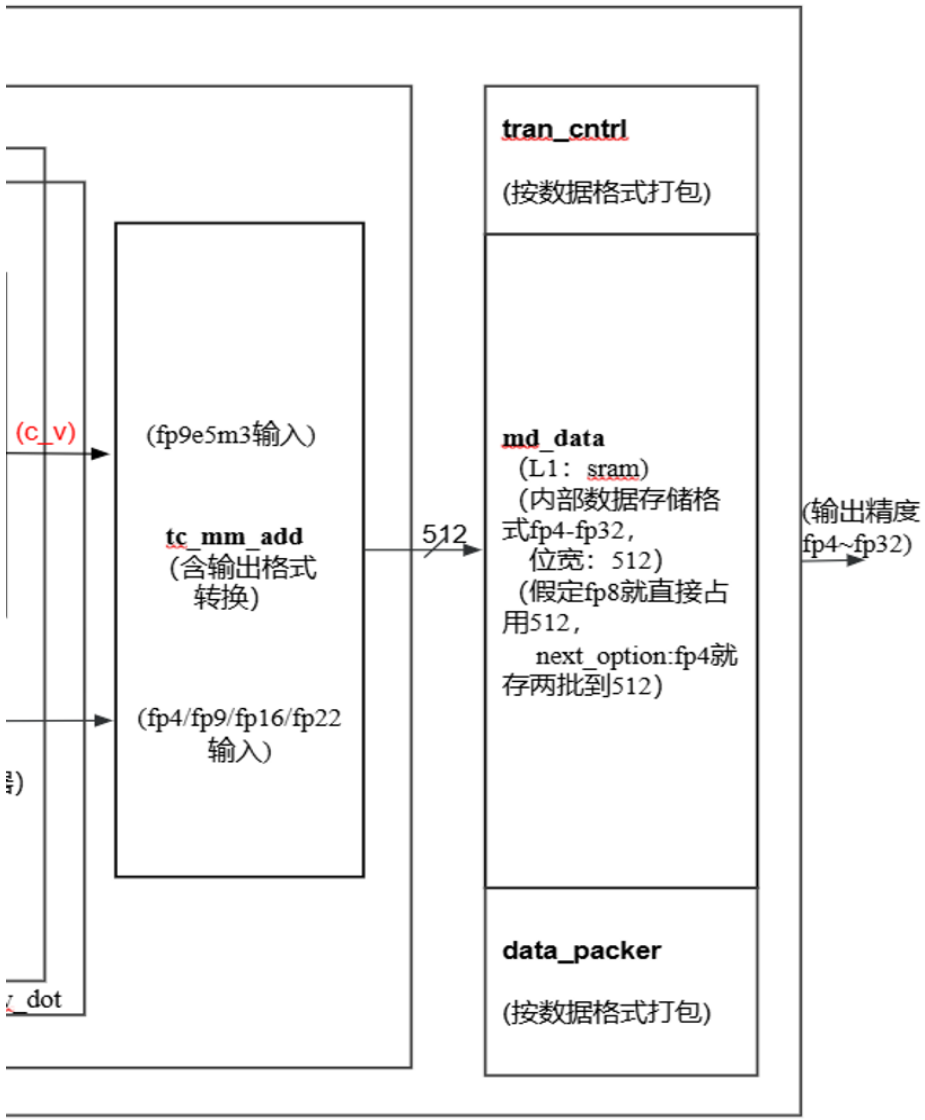
输出打包模块



输出打包模块

- tc_mm_add模块
  - ✓ 对前级模块mv_mul的输出（c_v）进行位拼接形成矩阵
  - ✓ 混合精度相加时，需要适当扩位
- md_data模块
  - ✓ 暂存tc_mm_add 输出结果的接受和缓冲区
  - ✓ 后续会由 tran_cntrl 和 data_packer 读取，封装输出
  - ✓ 对齐写入机制
    - ➢ 内部 SRAM 是按 fp9 元素宽度组织的
    - ➢ fp4 两拍数据合并后才能写一次
    - ➢ fp16 两个周期收够数据再写一次

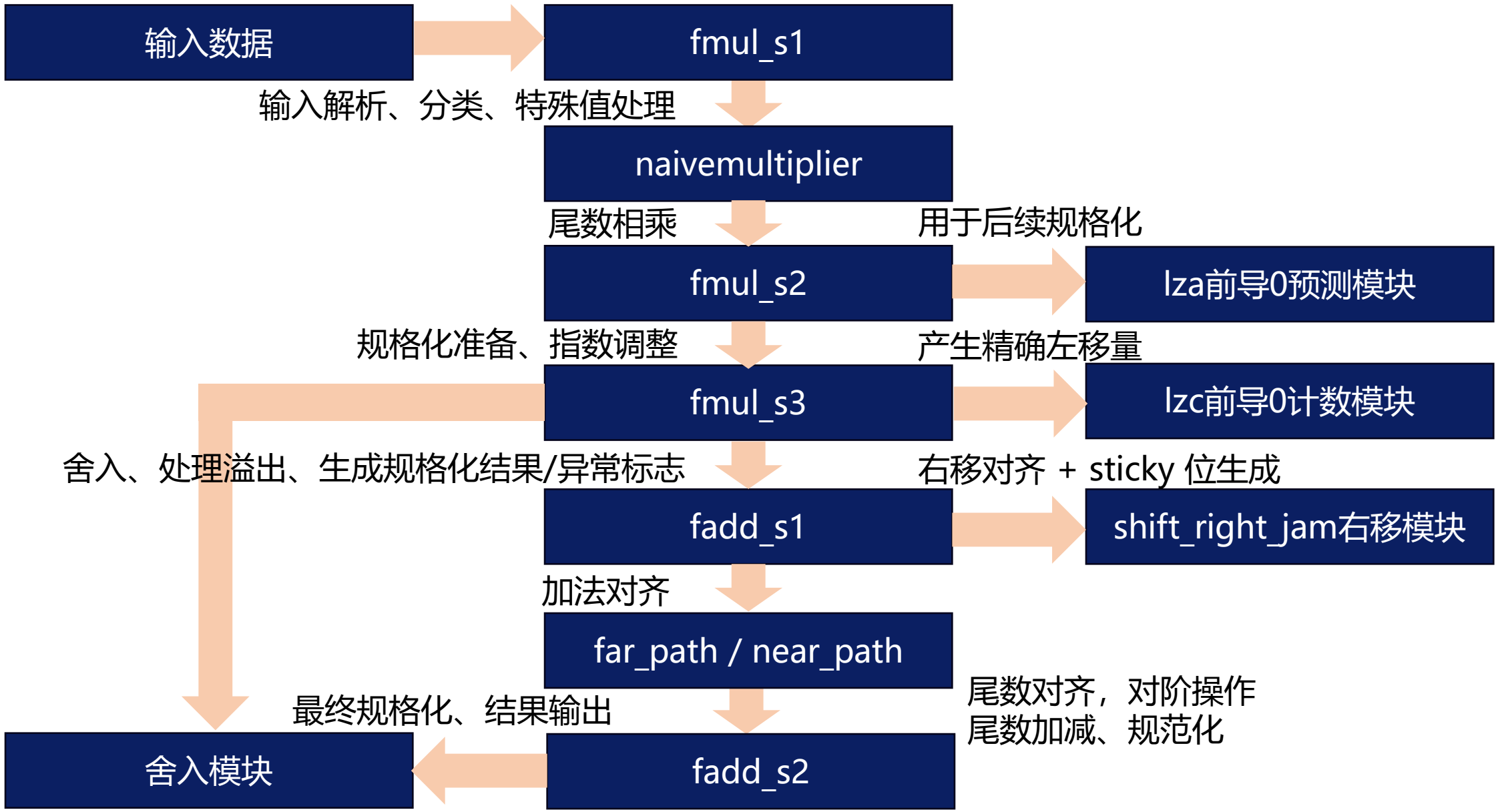
输出打包模块



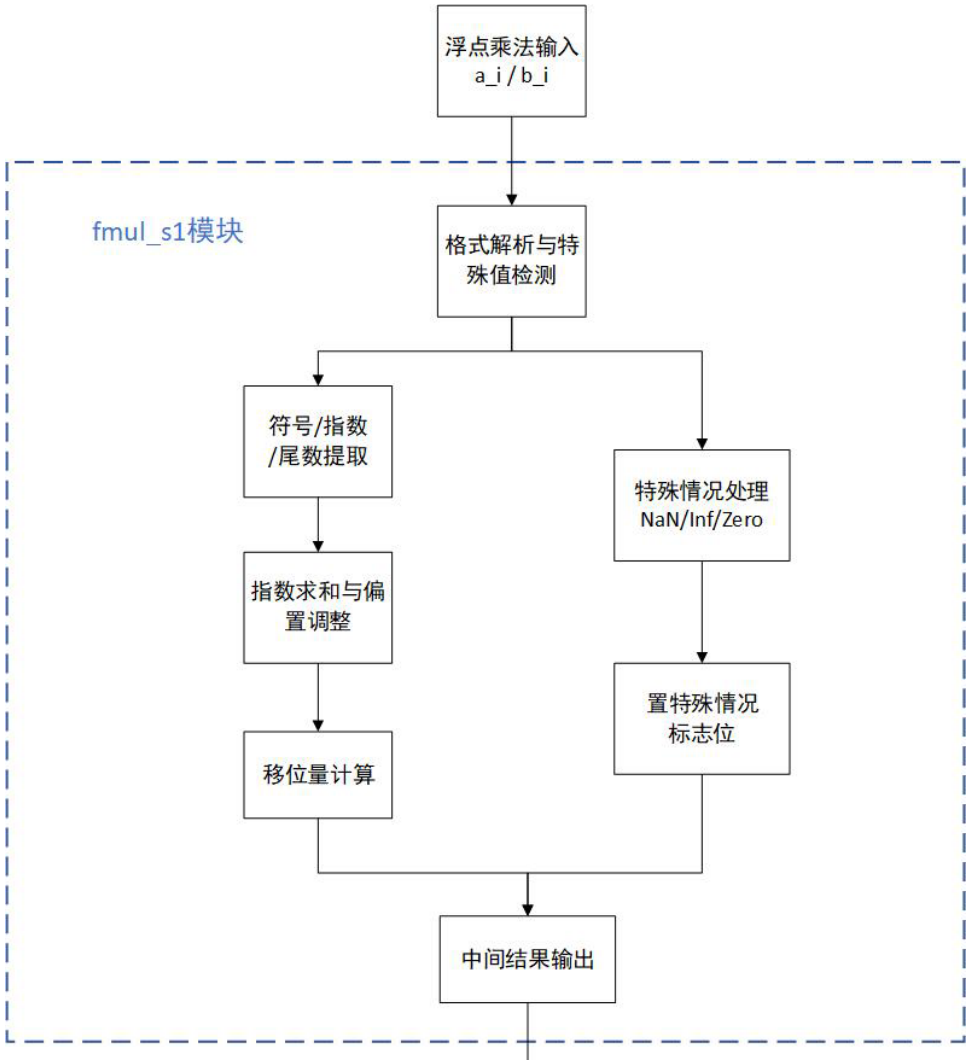
输出打包模块

- data_packer
  - ✓ 把整理好的数据封装成标准化包
- tran_cntrl
  - ✓ 按照目标数据格式打包
  - ✓ 保证外部接口协议（如 AXI4-Stream）对齐
  - ✓ 按输出目标（fp4、fp8、fp16、fp32）动态调度数据流

基本运算模块流程图


输入数据
fmul_s1
输入解析、分类、特殊值处理
naivemultiplier
尾数相乘
用于后续规格化
fmul_s2
lza前导0预测模块
规格化准备、指数调整
产生精确左移量
fmul_s3
lzc前导0计数模块
舍入、处理溢出、生成规格化结果/异常标志
右移对齐 + sticky 位生成
fadd_s1
shift_right_jam右移模块
加法对齐
far_path / near_path
尾数对齐，对阶操作
尾数加减、规范化
最终规格化、结果输出
舍入模块
fadd_s2

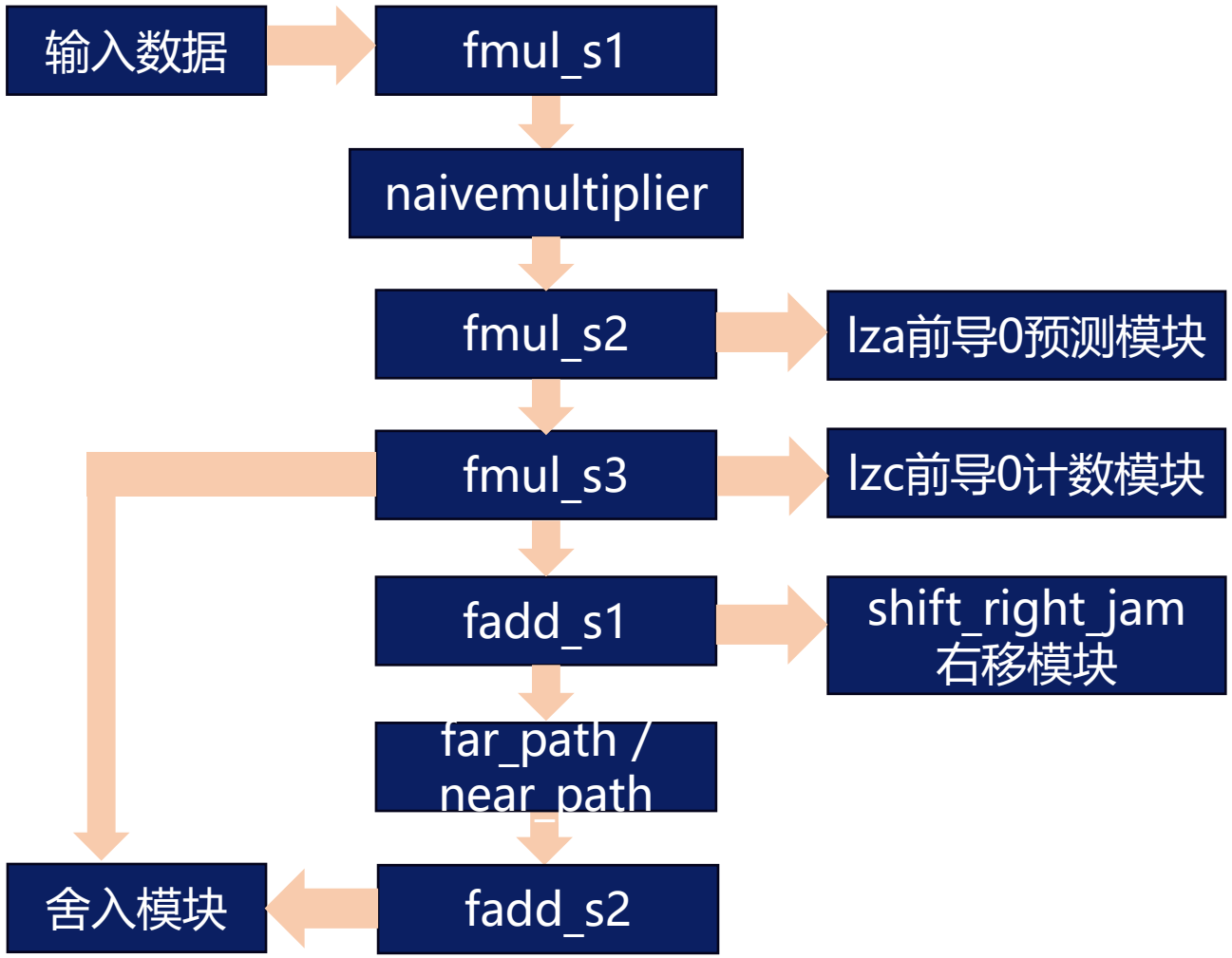
基本运算模块



□ fmul_s1模块

- ➢ **位置**：乘法单元 tc_mul 的第 1 阶段
- ➢ **作用**：输入解析 + 分类 + 初步乘法准备
- ✓ 对输入浮点数进行分类（正规化/非正规化/NaN/Inf）
- ✓ 拆解符号、指数、尾数
- ✓ 处理特殊情况（NaN、Inf、0）
- ✓ 准备规格化参数送给后续流水线
- ✓ 它是 乘法流水线的“入口”模块，所有 A/B 矩阵元素进入计算阵列后都先经过这里。

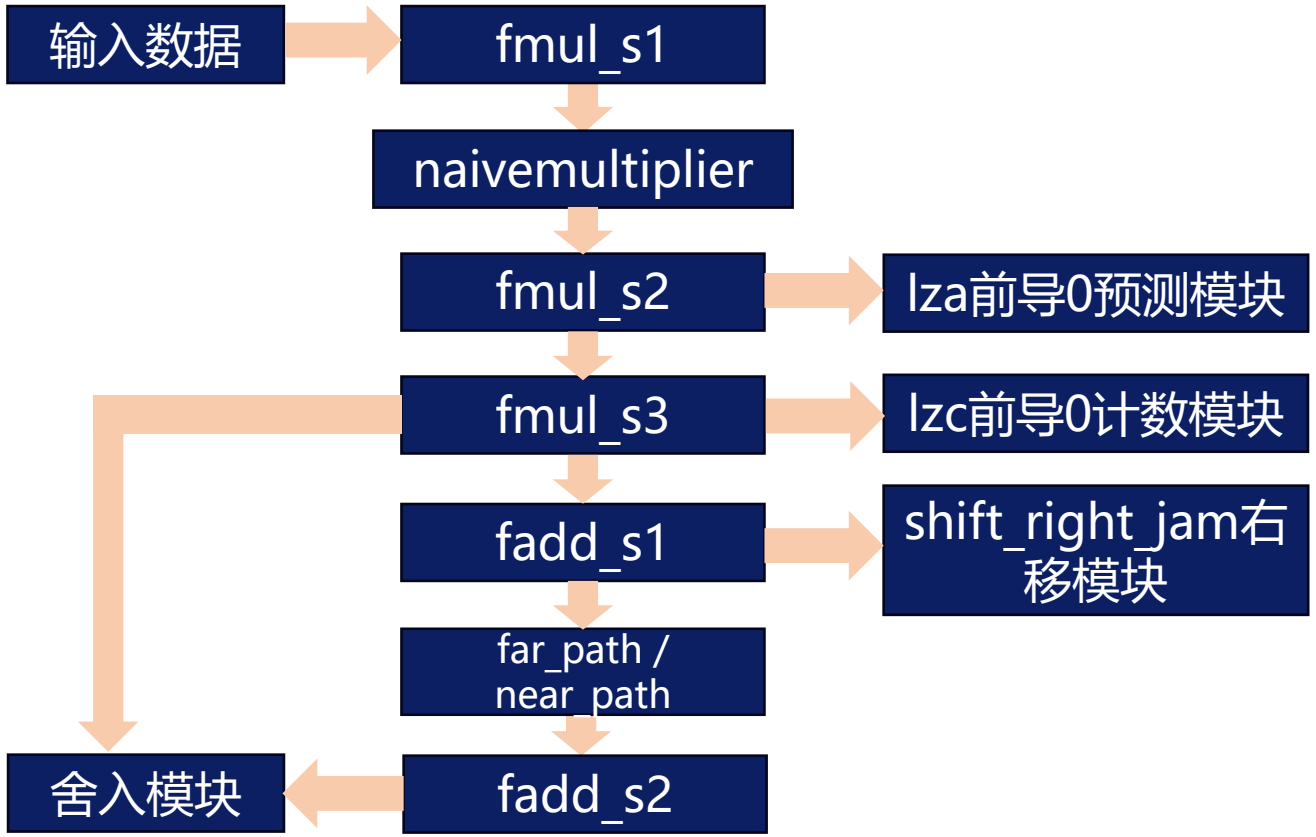
基本运算模块

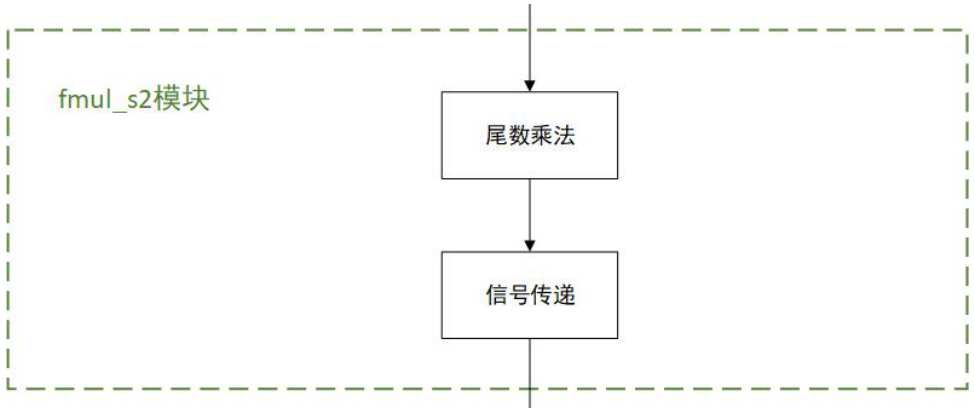


□ naivemultiplier 模块

- ➢ **位置**：fmul_s1 之后
- ➢ **作用**：尾数乘法器
- ✓ LEN=4 表示尾数宽度为 4 bit
- ✓ 输入：矩阵 A、B 的尾数部分
- ✓ 输出：尾数乘积结果（长度翻倍 LEN×2）
- ✓ 它是整个乘法过程中执行真正乘法的底层算术单元，对应 A × B

基本运算模块

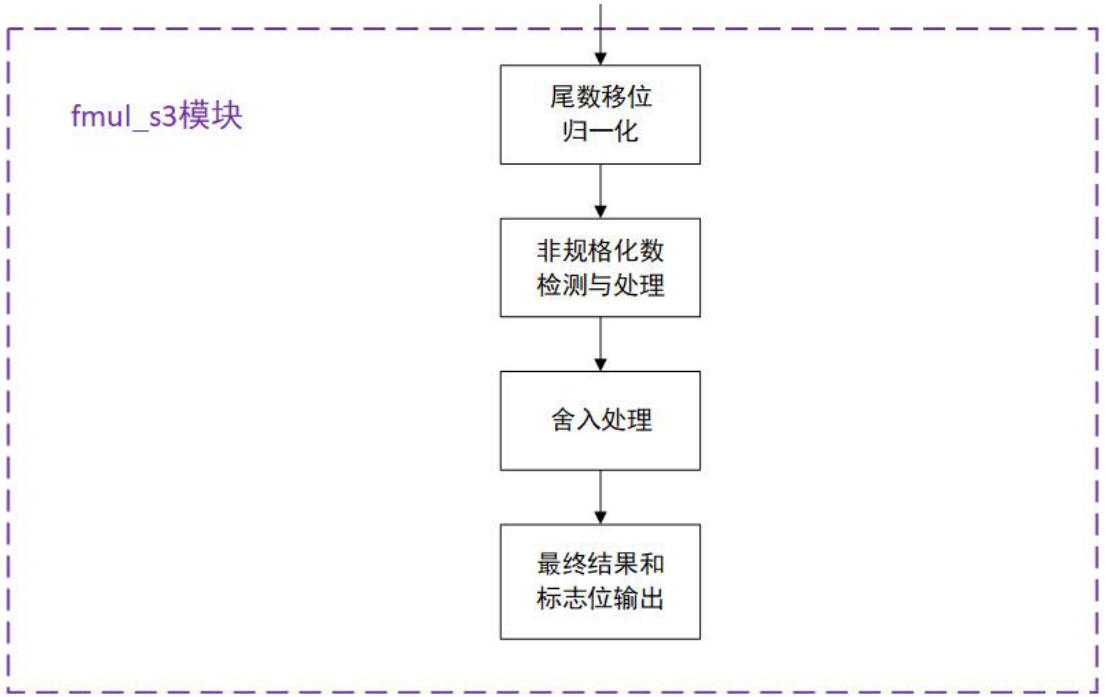




◘ fmul_s2 模块

- ➢ **位置**：乘法流水线第 2 阶段
- ➢ **作用**：规格化、边界情况处理
- ✓ 接收 S1 阶段拆分的符号、指数、尾数乘积
- ✓ 对指数进行偏移调整
- ✓ 准备规格化所需的中间参数
- ✓ 它是乘法的“中间处理”层，确保后续的规格化、舍入等操作可以正确执行

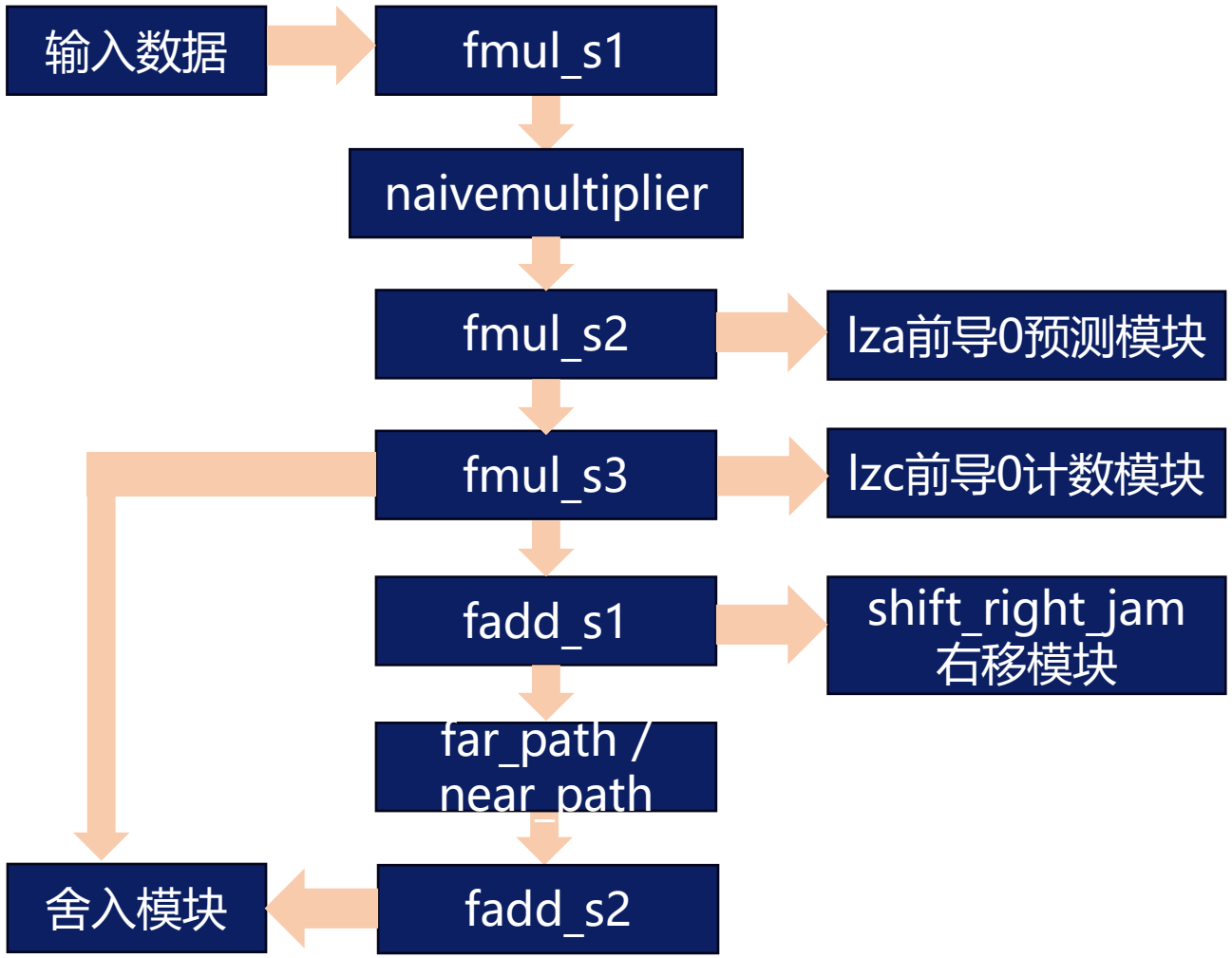
基本运算模块



◘ fmul_s3 模块

- ➢ **位置**：乘法流水线第 3 阶段（出口阶段）
- ➢ **作用**：最终规格化 + 溢出处理 + 输出结果
- ✓ 对乘积结果进行最终规格化（normalize）
- ✓ 检查并处理溢出/下溢
- ✓ 生成 IEEE 754 异常标志
- ✓ 把结果送入加法树（即 tc_add）
- ✓ 它是整个乘法运算的“出口”，把乘法结果包装成标准浮点格式供后续累加使用

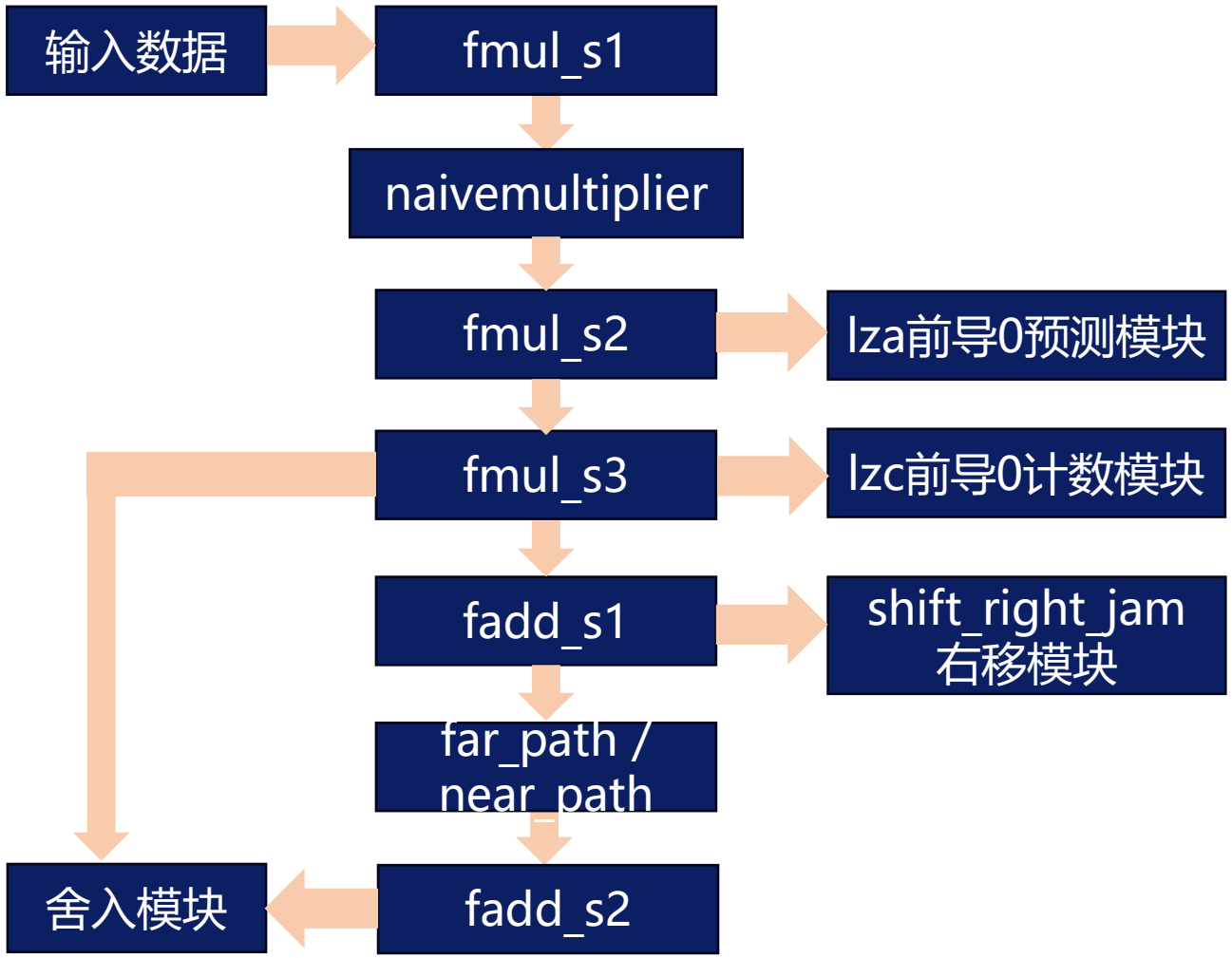
基本运算模块



□ fadd_s1 模块

- ➢ **位置**：加法单元 tc_add 的第 1 阶段
- ➢ **作用**：加法对齐阶段（对阶 + 尾数对齐）
- ✓ 根据指数差调整对齐
- ✓ 控制尾数移位（准备相加/相减）
- ✓ 为远路径/近路径选择做准备
- ✓ 这是浮点加法/累加的“入口模块”，接收来自乘法器的结果

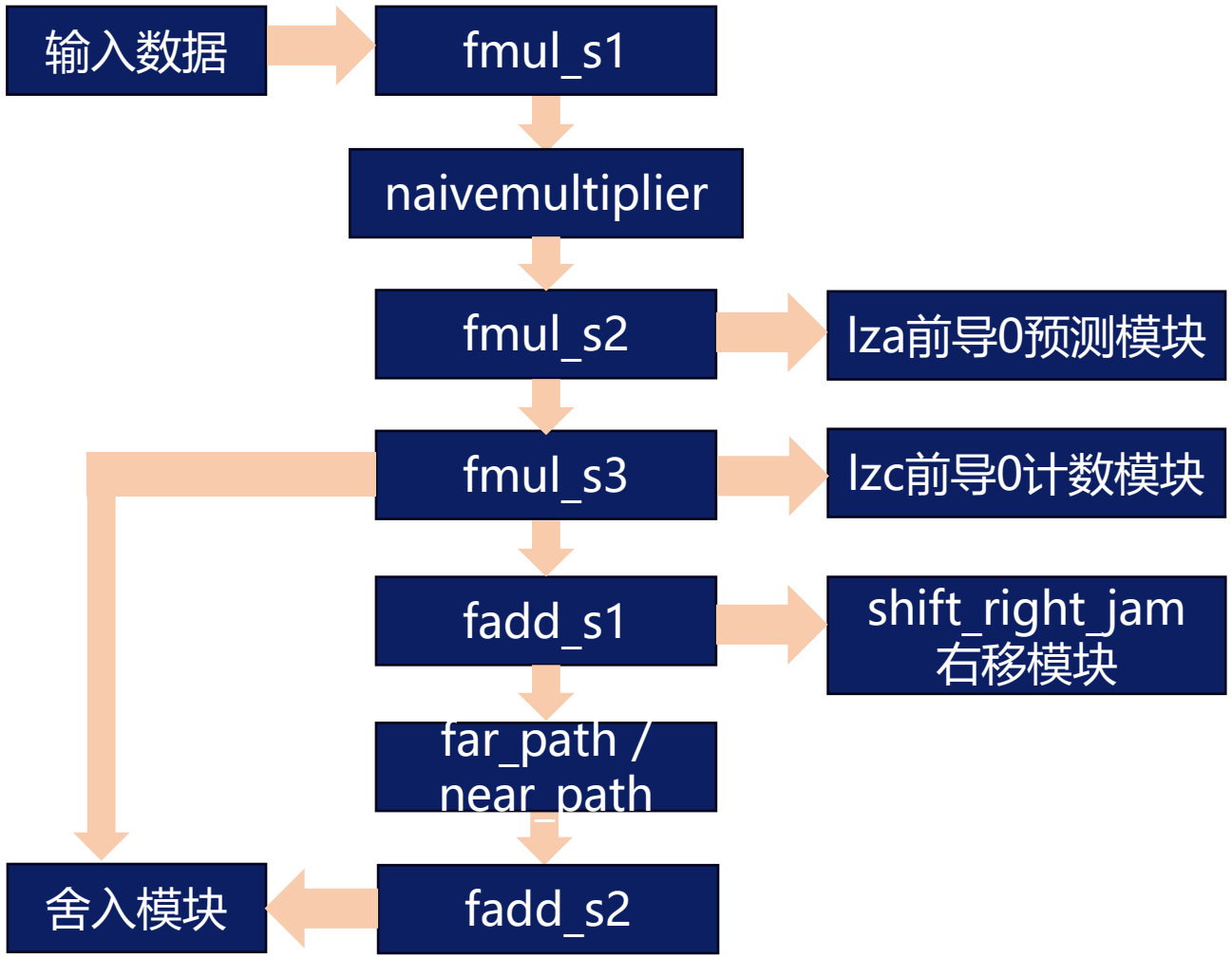
基本运算模块



far_path 模块

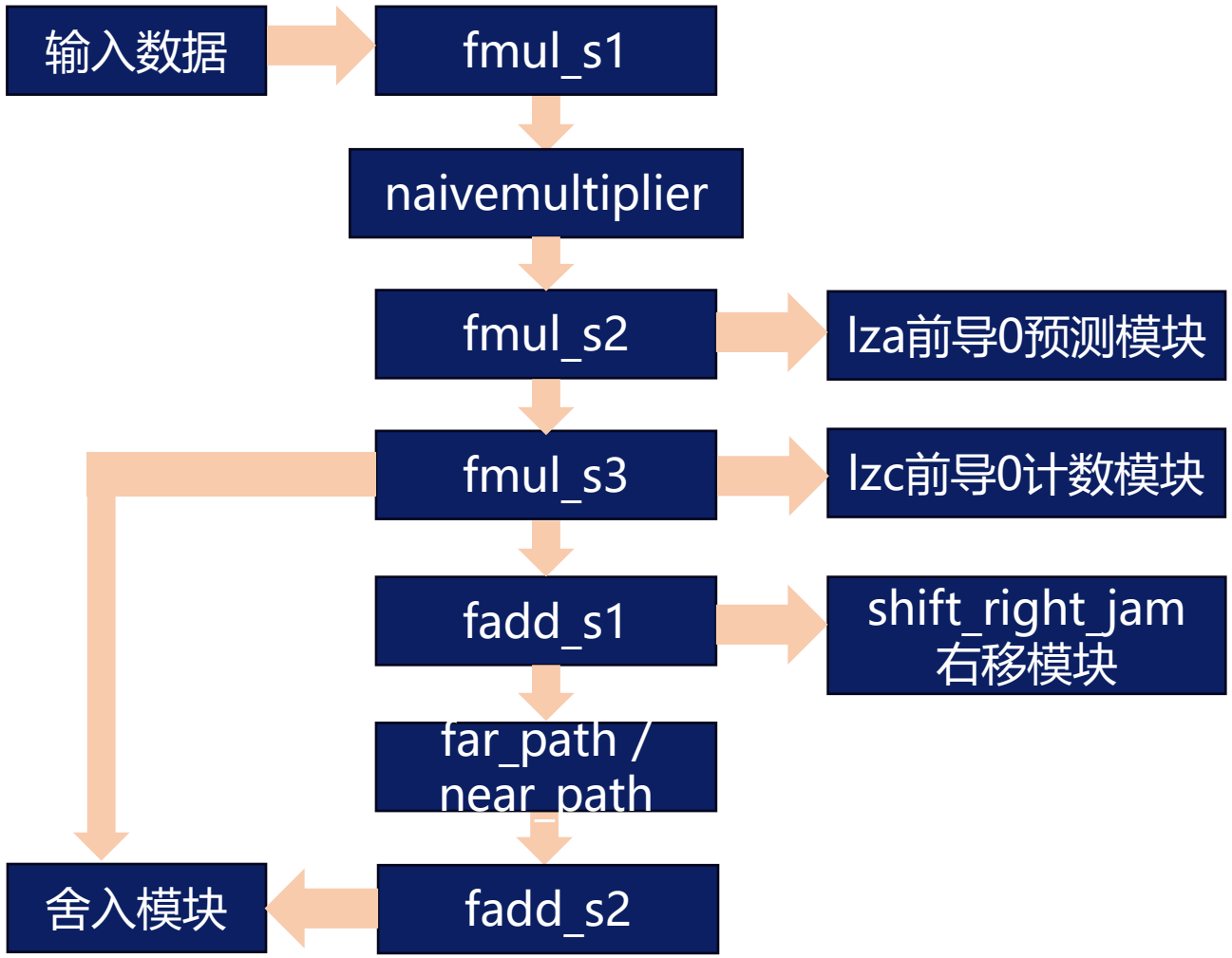
- **位置**：加法路径中指数差较大（≥2）的分支
- **作用**：远路径加法逻辑
- ✓ 使用 shift_right_jam 对尾数对齐
- ✓ 处理尾数加/减运算（带符号相加/补码）
- ✓ 根据加法结果判断是否规格化、是否进位等
- ✓ 这是浮点加法器中专门处理“大指数差”情况的路径，是高精度对齐的关键部分

基本运算模块



- **near_path 模块**
  - **位置**：加法路径中指数差较小（<2）的分支
  - **作用**：近路径加法逻辑
  - ✓ 实现小指数差时的快速加法运算
  - ✓ 无需大范围尾数对齐，延迟更低
  - ✓ 和 far_path 互补，它专门优化“指数接近”的情况，使加法器更快

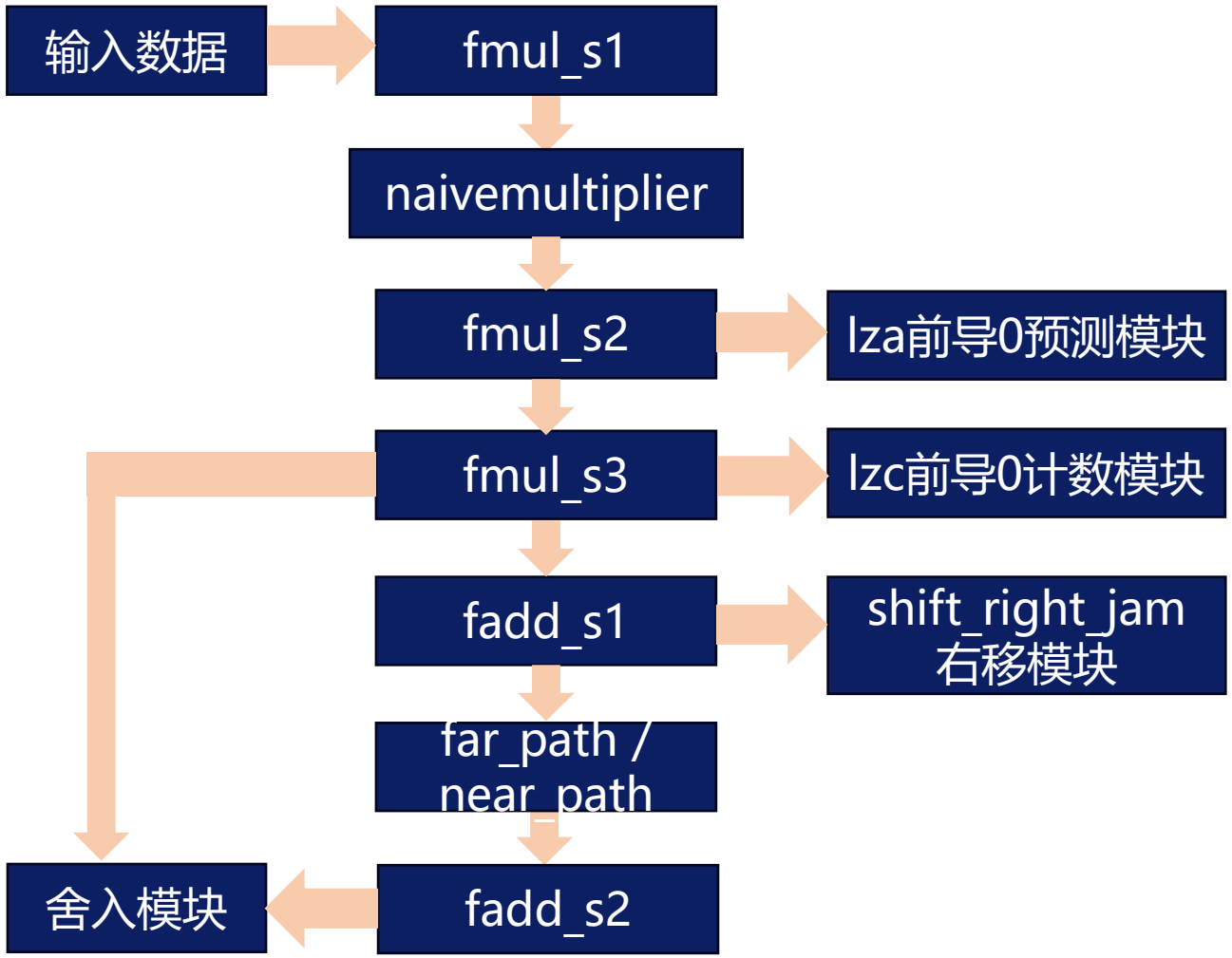
基本运算模块



◘ fadd_s2 模块

- ➢ **位置**：加法单元最终阶段
- ➢ **作用**：规格化 + 溢出处理 + 输出结果
- ✓ 对第一阶段的加法结果进行规格化
- ✓ 处理溢出/下溢
- ✓ 输出最终结果及异常标志
- ✓ 它是整个加法器的“出口”，把加法结果变成标准浮点输出

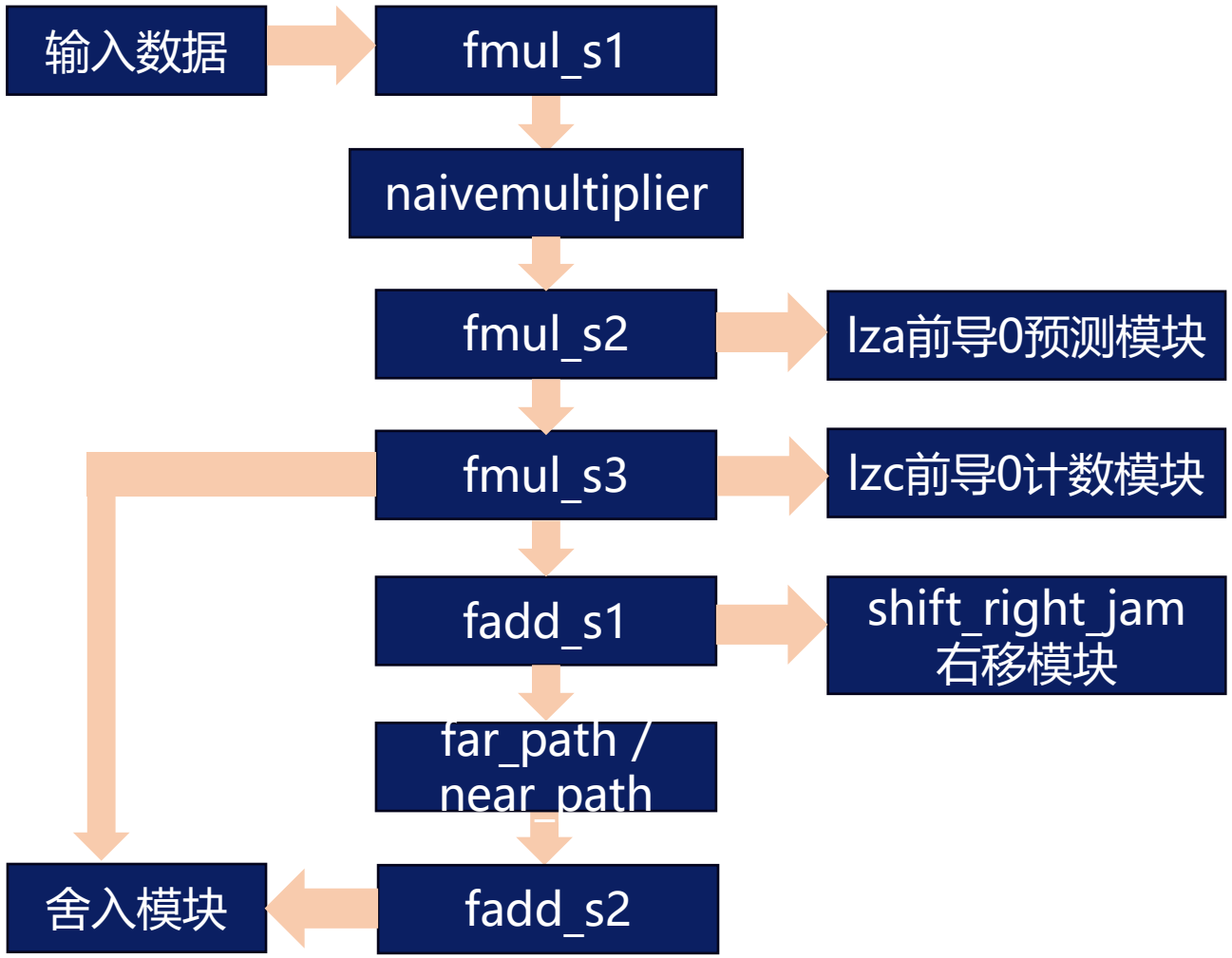
基本运算模块



舍入模块

- ✓ **位置：** 浮点运算流水线的最后阶段
- ✓ **作用：** 根据 IEEE 754 舍入模式（RNE、RTZ、RUP、RDN、RMM），对尾数做最后一位的进位与截断处理

基本运算模块



□ lza 前导 0 预测模块（Leading Zero Anticipate)

✓ **位置：** 浮点规格化阶段的前置逻辑

✓ **作用：** 提前预测运算结果中前导零的位置，加快归一化过程

➢ 为了加快速度，lza 模块提前预测第一个非零位的位置

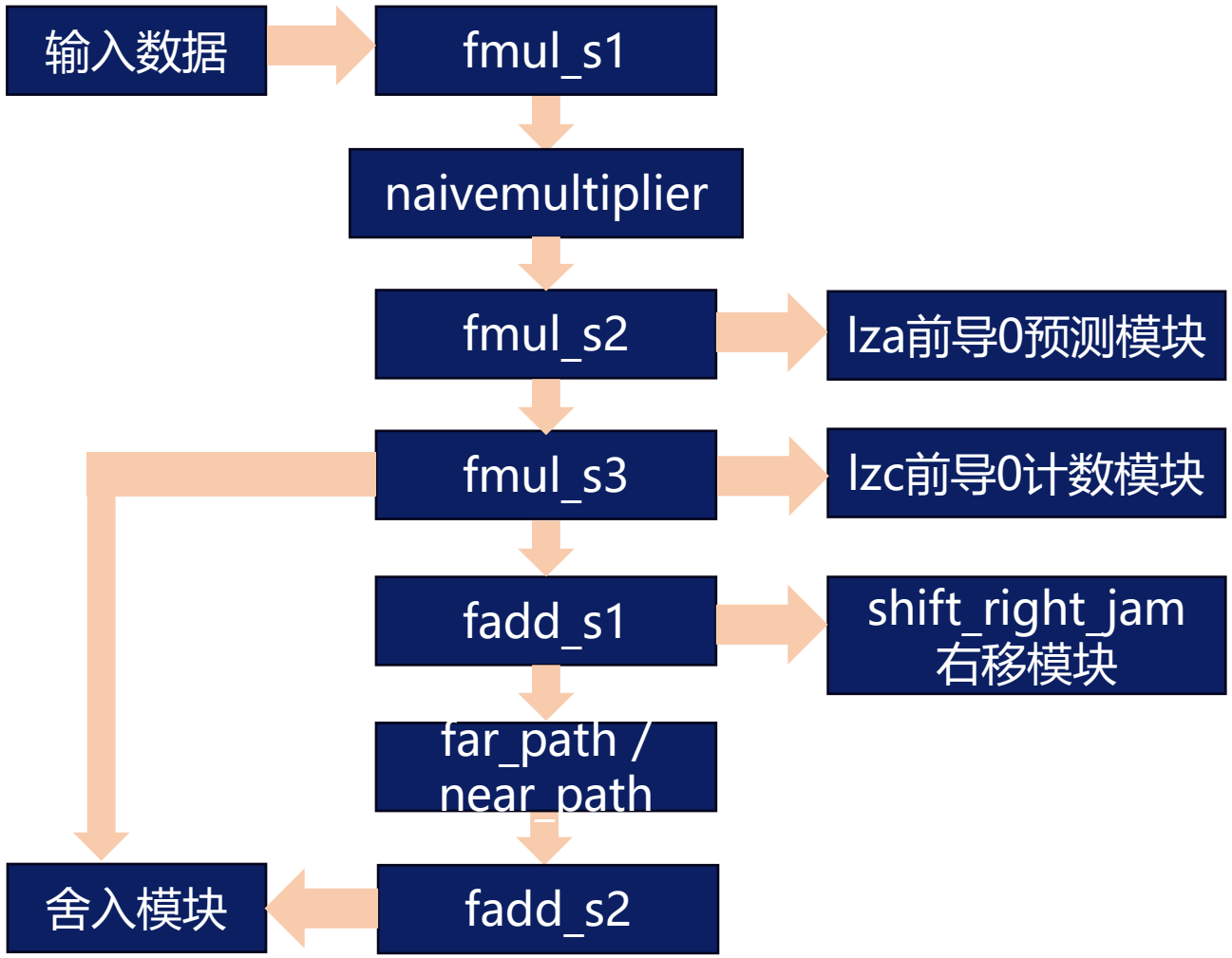
➢ 它是一种“快速估算”逻辑，不需要完全展开计算结果，就能预测出前导零数

□ lzc 前导 0 计数模块（Leading Zero Counter)

✓ **位置：** 紧跟在 lza 模块后面，用于精确规格

✓ **作用：** 根据 lza 的预测结果，精确计算尾数中前导零的数量

基本运算模块



□ shift_right_jam 模块

✓ **位置：** 浮点加法的“对阶阶段”（`fadd_s1 / far_path`）

✓ **作用：** 实现右移并检测黏着位是否为1

- ➢ 当两个浮点数指数不同，较小的那个尾数要右移对齐
- ➢ 但右移时会丢弃低位，这些被丢弃的位会影响舍入判断
- ➢ 因此引入 sticky 位（粘滞位），表示“移出去的低位是否包含 1”

未来改进方向

- 在兼容gpgpu-sim的前提下**支持FP8和FP4**
- 未来整合入**vortex**
- **3D/3.5D**架构的TensorCore

课程基本信息

- 课程名称：基于RISC-V的开源GPU架构与设计探索
- 授课地点：深圳市学苑大道南山智园C3栋20层2005（清华大学深圳国际研究生院）
- 授课时间：十周（每周六，9:00-12:00 14:00-18:00）
- 本课程聚焦RISC-V开源架构与GPU设计的**热门交叉前沿**，构建从指令集到众核计算架构的**探索性知识体系**，以**最短路径**实现RISC-V GPU体系架构入门，并探讨3D/3.5D Chiplet、开源EDA等技术在开源GPU中的应用潜力
- 课程核心内容涵盖：RISC-V指令集架构（ISA）与GPU并行计算架构的融合设计，开源GPU核心模块（如流处理器、存储架构、TensorCore），核心编译器的原理与探索实现。通过C-Model仿真与FPGA实现，学生将掌握从架构建模、功能验证到部署实现的精简芯片设计流程，并探索开源RISC-V GPU的大模型（Transformer）应用，为进入AI芯片/GPU编程等前沿领域的研究生学习或职业发展奠定**核心竞争力与先发优势**
- https://github.com/chenweiphd/OpenRVGPUCourse