

github · tiny-tpu-main

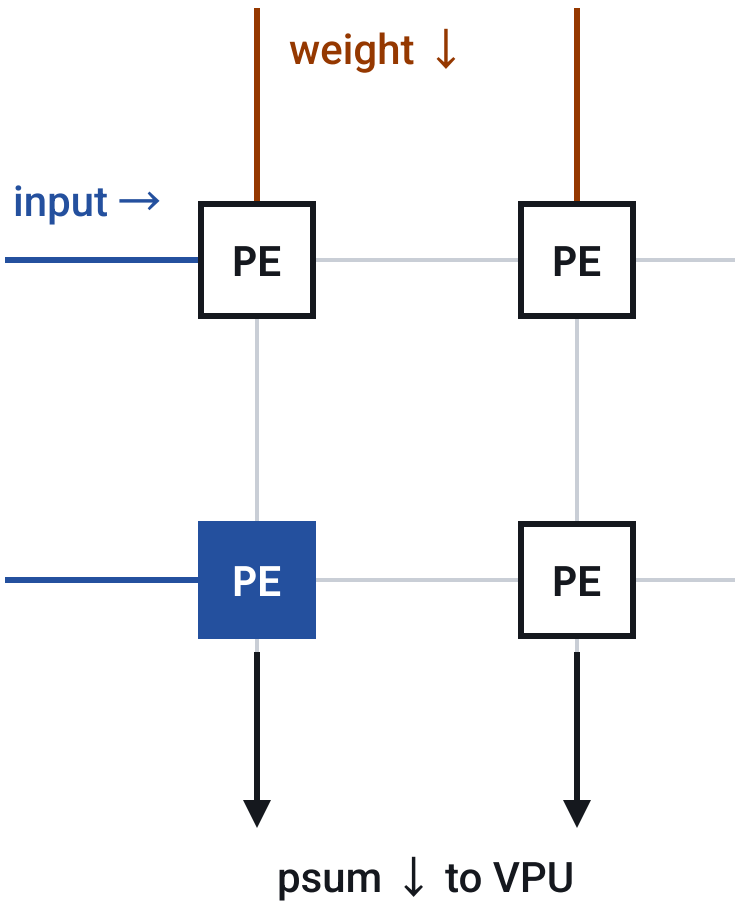
tiny-tpu

代码结构与核心实现

一个用 SystemVerilog 从零实现的最小化 Tensor Processing Unit，参考 Google TPU v1 / v2 重新设计——支持矩阵乘法、激活函数、MSE loss 与完整的反向传播训练。

A minimal tensor processing unit, reinvented from Google's TPU v1 & v2.

16	3,162	2×2	Q8.8
RTL modules (src/)	行 SystemVerilog	Systolic Array	16-bit fixed-point



这次走读覆盖什么

01	项目结构	Repository layout	02	顶层架构	tpu.sv top level
03	定点数库	fixedpoint.sv	04	处理单元 PE 核心	pe.sv
05	脉动阵列 核心	systolic.sv	06	向量处理单元 核心	vpu.sv
07	统一缓冲区 核心	unified_buffer.sv	08	指令集与控制单元 核心	ISA · control_unit.sv
09	验证与演示	cocotb · SVA · FPGA	10	设计要点回顾	Takeaways

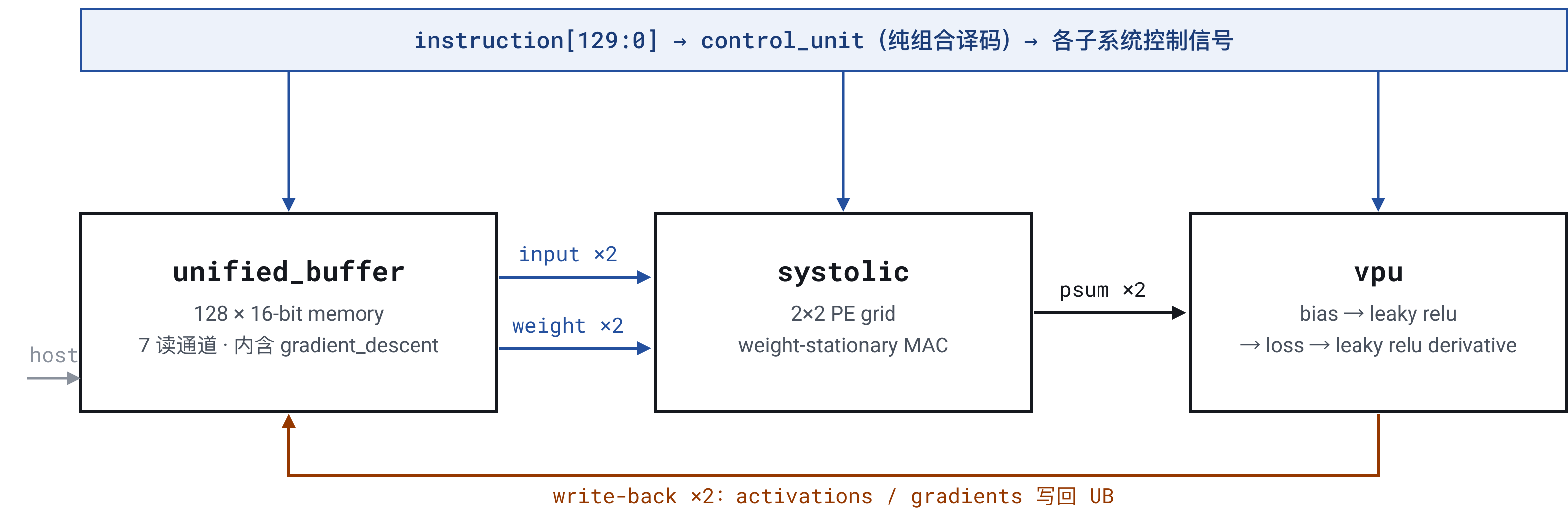
仓库结构：核心 RTL 只有 src/ 一个目录

tiny-tpu-main/	
└ src/	16 个 SystemVerilog 模块 · 核心 RTL (3,162 行)
└ sva/	SVA 断言 12 组 + bind 脚本 + 顶层 testbench
└ test/	cocotb 测试 (test_*.py) 与波形 dump 顶层 (dump_*.sv)
└ docs/	验证计划、formal 覆盖率与结果报告
└ jupyter/	NumPy golden model (single_pass_numpy.ipynb)
└ mnist_demo/	DE1-SoC FPGA 演示: MNIST 手写数字训练/推理
└ xor_demo/	DE1-SoC FPGA 最小演示: XOR 训练闭环
└ tiny-tpu-hardened/	ASIC 加固版 RTL + OpenLane 物理设计输入
└ images/ videos/	架构图与演示录像
└ Makefile	cocotb + iverilog 仿真入口 (每个模块一条 target)

- **src/ 是全部核心**: 16 个模块、3,162 行, 本次走读的主角; 模块间只靠端口连接, 无全局状态
- **验证与实现分离**: sva/ 放断言、test/ 放 cocotb、docs/ 放 formal 报告, RTL 目录保持干净
- **同一套 RTL 三种去向**: 仿真 (test/)、FPGA (mnist_demo / xor_demo)、ASIC (tiny-tpu-hardened 的 OpenLane flow)

读法建议 自底向上: fixedpoint → pe → systolic → vpu → unified_buffer → tpu, 最后看 ISA 如何把整台机器串起来。

tpu.sv: 三个子系统组成一条闭环数据通路



- 统一数据格式:** 全通路 16-bit signed fixed-point (Q8.8) , 每个模块自带 sticky overflow 上报
- 前向 / 反向同一条环:** UB → Systolic → VPU → UB; 走哪几级由 `vpu_data_pathway[3:0]` 切换
- 控制即指令:** control_unit 把 130-bit instruction 逐字段 assign 到控制线, 无状态机、无寄存器

fixedpoint.sv: 全部算术的地基 (Q8.8)

src/fixedpoint.sv · fxp_mul1,250 行库

```
module fxp_mul #(
    parameter WIIA = 8,    // 整数位宽
    parameter WIFA = 8,    // 小数位宽 → Q8.8
)(
    input wire [WIIA+WIFA-1:0] ina, // 16-bit signed
    input wire [WIIB+WIFB-1:0] inb,
    output wire [WOI +WOF -1:0] out,
    output wire overflow
);
wire signed [WRI+WRF-1:0] res = $signed(ina) * $signed(inb);

fxp_zoom #(.WII(WRI), .WIF(WRF), .WOI(WOI),
           .WOF(WOF), .ROUND(ROUND))
    res_zoom (.in(res), .out(out), .overflow(overflow));
```

Q8.8: 范围 ± 127.996 · 分辨率 $2^{-8} \approx 0.0039$

module	功能	时序
fxp_zoom	位宽转换 + 舍入	组合
fxp_add / fxp_addsub	加 / 加减	组合
fxp_mul	乘法（全精度乘 + zoom 截位）	组合
fxp_mul_pipe	乘法	2 级流水
fxp_div / fxp_sqrt	除 / 开方（critical path 长）	组合
fxp2float / float2fxp	定点 $\leftrightarrow$ 浮点（调试用）	组合 / 流水

设计意图 乘积先以全精度（WRI+WRF 位）保存，再经 fxp_zoom 统一舍入并产生 overflow——舍入策略集中在一处，所有算子共享。

PE：每周期一次 Multiply-Accumulate

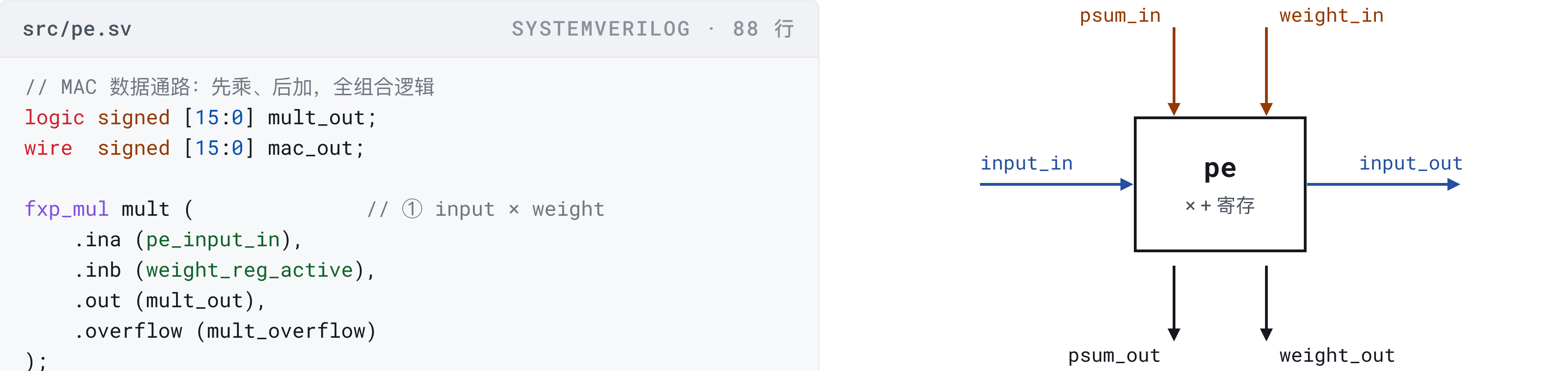
src/pe.svSYSTEMVERILOG · 88 行

```
// MAC 数据通路：先乘、后加，全组合逻辑
logic signed [15:0] mult_out;
wire signed [15:0] mac_out;

fxp_mul mult (                // ① input × weight
    .ina (pe_input_in),
    .inb (weight_reg_active),
    .out (mult_out),
    .overflow (mult_overflow)
);

fxp_add adder (                // ② 乘积 + 上方传来的 psum
    .ina (mult_out),
    .inb (pe_psum_in),
    .out (mac_out),
    .overflow (add_overflow)
);
```

psum_out = input_in × weight + psum_in



控制信号（西侧随路）：valid_in · switch_in · enabled

- **Weight-stationary 数据流**：weight 常驻 PE 内部寄存器，activation 水平流动、psum 垂直累积
- **四向端口**：北入（psum、weight）、西入（input）、南出（psum、weight 传递给下一行）、东出（input 传递给右邻）
- **Sticky overflow**：溢出标志一旦置位就保持，供上层观测

PE 的 always_ff: weight 双缓冲与控制波

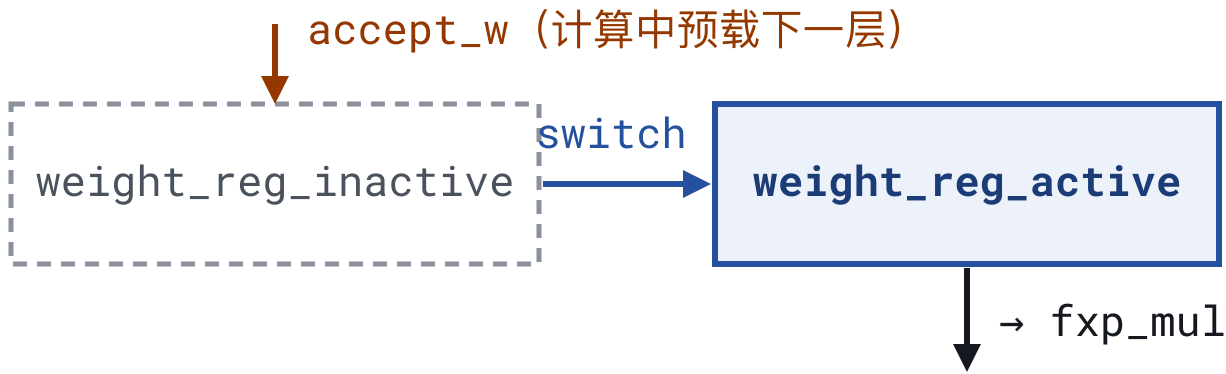
src/pe.sv · always_ff

唯一的时序块

```
if (pe_switch_in) // ① 换层: shadow → active
    weight_reg_active <= weight_reg_inactive;

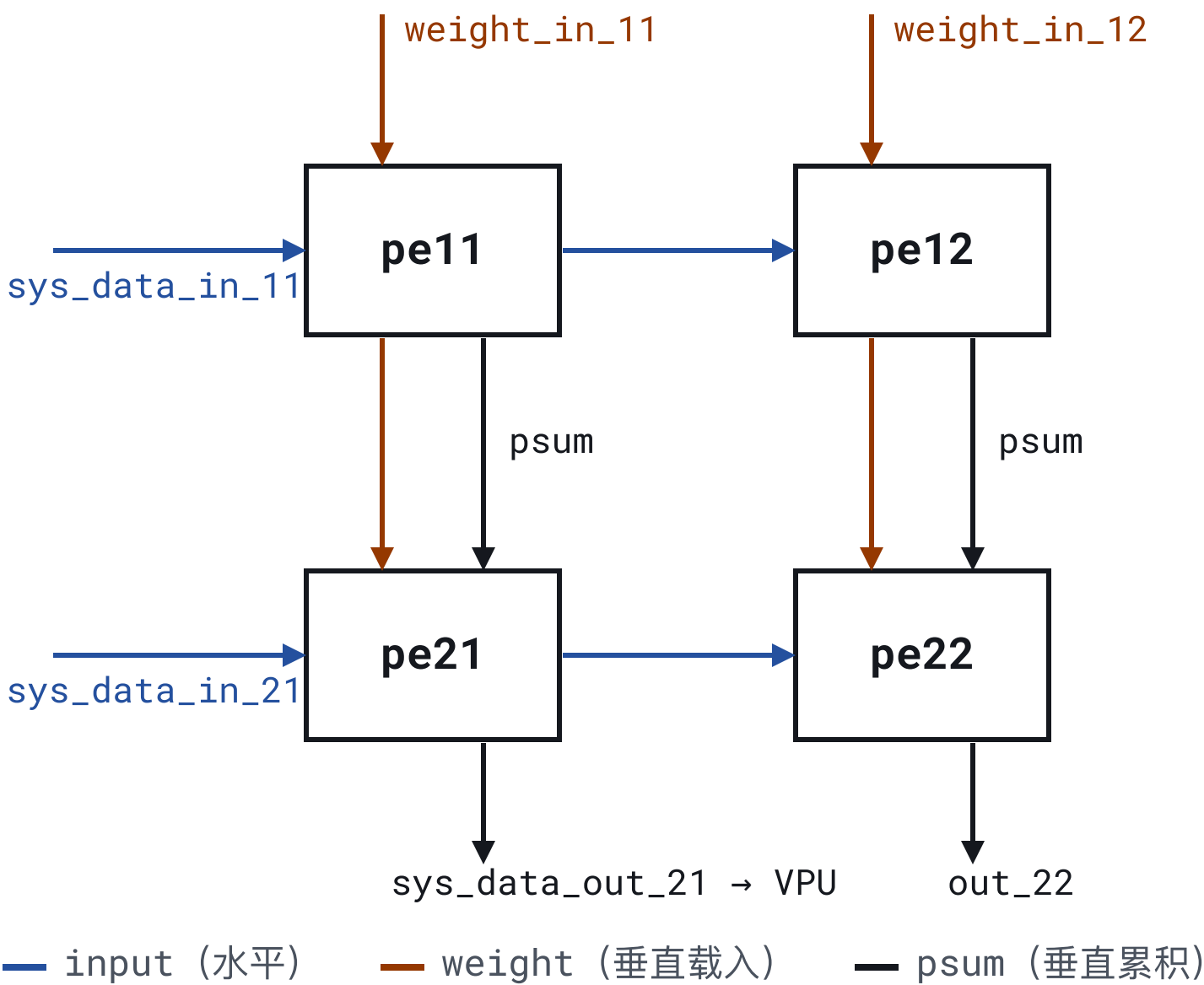
if (pe_accept_w_in) begin // ② 载入 weight: 先入 shadow
    weight_reg_inactive <= pe_weight_in;
    pe_weight_out <= pe_weight_in; // 同时向南接力
end else
    pe_weight_out <= 0;

if (pe_valid_in) begin // ③ 数据流: 打一拍输出
    pe_input_out <= pe_input_in; // activation → 东邻
    pe_psum_out <= mac_out; // psum → 南邻
    pe_overflow_out <= pe_overflow_out
        | mult_overflow | add_overflow; // sticky
end
```



- **双缓冲换层**: 当前层计算的同时把下一层 weight 装进 shadow 寄存器，一个 switch 脉冲全阵切换
- **控制波传播**: valid / switch 每拍向东、向南接力，天然形成 systolic 所需的 stagger 时序
- **列屏蔽**: rst 或 pe_enabled=0 时全部输出清零，被禁用列不污染 psum

systolic sv: 2×2 阵列是「画」出来的连线



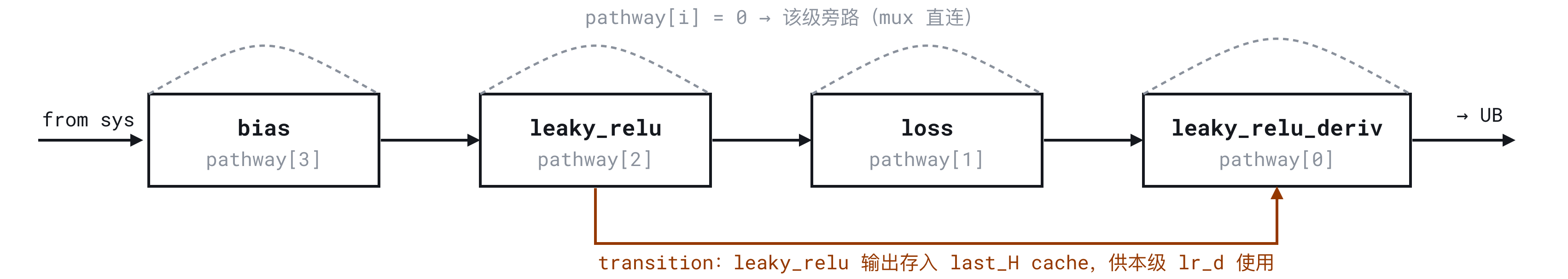
src/systolic sv · 列屏蔽

154 行

```
// 只启用矩阵实际需要的列
always_ff @(posedge clk or posedge rst) begin
    if (rst) pe_enabled <= 2'b11;
    else if (ub_rd_col_size_valid_in)
        case (ub_rd_col_size_in[1:0])
            2'd1: pe_enabled <= 2'b01; // 1 列
            2'd2: pe_enabled <= 2'b11; // 2 列
            default: pe_enabled <= 2'b00;
        endcase
    end
```

- **显式例化**: pe11/pe12/pe21/pe22 四个实例手工连线，没有 generate 循环——2×2 是画出来的，SYSTOLIC_ARRAY_WIDTH 参数为扩展预留
- **数据节奏**: 两行 input 由独立的 sys_start_1/2 错峰打入；psum 沿列向下累积，底行输出即矩阵乘结果
- **列屏蔽**: UB 读命令携带 col_size，阵列据此禁用多余列，省电且不污染结果

vpu.sv：四级算子链，用 4-bit pathway 现场接线



vpu_data_pathway	阶段	实际通路
4'b1100	forward（隐藏层）	sys → bias → leaky relu → UB
4'b1111	transition（输出层）	sys → bias → lr → loss → lr_d → UB（缓存 H）
4'b0001	backward（反向）	sys → lr_d（H 取自 UB）→ UB
4'b0000	idle	全部旁路，模块输入清零

- **一位一级**：pathway 的每个 bit 控制一级使能；禁用级输入清零、数据旁路直穿
- **H 矩阵两级来源**：transition 当拍用 last_H cache；之后各层反向从 UB 读 H
- **输出打拍**：vpu_data_out 经寄存器输出，消除 mux 组合毛刺

四个算子 child 模块：每个只有几十行

leaky_relu_child

forward · 45 行

$$f(x) = x \geq 0 ? x : \alpha \cdot x$$

```
if (lr_data_in >= 0)
    lr_data_out <= lr_data_in;      // 正区直通
else
    lr_data_out <= mul_out;         // 负区 × leak factor α
```

leaky_relu_derivative_child

backward · 46 行

$$dH = H \geq 0 ? d : \alpha \cdot d$$

```
if (lr_d_H_data_in >= 0)    // 用 H 的符号选分支
    lr_d_data_out <= lr_d_data_in;
else
    lr_d_data_out <= mul_out;    // 上游梯度 × α
```

loss_child

MSE 梯度 · 62 行

$$\partial L / \partial H = (2/N) \cdot (H - Y)$$

```
fxp_addsub subtractor (.ina(H_in), .inb(Y_in),
    .sub(1'b1), .out(diff_stage1), ...);    // stage 1
fxp_mul multiplier (.ina(diff_stage1),
    .inb(inv_batch_size_times_two_in), ...); // stage 2
```

gradient_descent

权重更新 · 位于 UB 内 · 82 行

$$w \leftarrow w - lr \cdot g$$

```
fxp_mul mul_inst (.ina(grad_in), .inb(lr_in), ...);
fxp_addsub sub_inst (.ina(sub_in_a), .inb(mul_out),
    .sub(1'b1), .out(sub_value_out), ...);
// bias 走累加路径, weight 一次更新 (grad_bias_or_weight)
```

unified_buffer: 一块内存 + 7 条独立读通道

- **单一存储**: `ub_memory[0:127]` (16-bit/字), `X`、`W`、`bias`、`Y`、`H`、梯度全部按地址约定共存
- **读模型**: 每条通道自带 `ptr / row_size / col_size / time_counter`; `ub_rd_start` 后每拍连续吐数, 读够计数自动停
- **transpose 在硬件做**: 启动读时交换 `row/col size`, 实现输入矩阵 90° 旋转与转置
- **训练闭环在片内**: `gradient_descent` 例化于 UB 内, 梯度算完就地写回 memory 完成权重更新

ub_ptr_select	读通道去向
0	input 矩阵 → systolic 左侧
1	weight 矩阵 → systolic 顶部 (stagger 地址)
2 / 3 / 4	bias / Y 标签 / H 激活 → VPU 各算子
5 / 6	bias 梯度 / weight 梯度 → gradient_descent

src/unified_buffer.sv · 读命令分发553 行

```
if (ub_rd_start_in) begin
  case (ub_ptr_select)
    0: rd_input_ptr  <= ub_rd_addr_in;
    1: rd_weight_ptr <= ub_rd_addr_in
      + ub_rd_row_size*ub_rd_col_size
      - ub_rd_col_size; // stagger
    2: rd_bias_ptr   <= ub_rd_addr_in;
    // 3: Y    4: H    5: grad_bias    6: grad_weight
  endcase
end
```

为什么叫 Unified 参照 Google TPU v1: 激活、权重、中间结果不分开存, 一块 SRAM 服务所有阶段——面积最小化, 代价是地址规划完全交给指令序列。

130-bit ISA：一条指令 = 一块硬件控制面板



src/control_unit.sv · 全部实现

67 行 · 纯组合

```
assign sys_switch_in    = instruction[0];
assign ub_rd_start_in   = instruction[1];
assign ub_rd_transpose  = instruction[2];
assign ub_ptr_select    = instruction[61:53];
assign vpu_data_pathway = instruction[97:94];
assign vpu_leak_factor_in = instruction[129:114];
// 没有状态机、没有寄存器：译码 = 位切片
```

- **宽指令 = 并行控制**：一条指令同时驱动 UB 读、host 写、VPU 通路和训练参数，每拍都可以换一条
- **版本差异**：README 描述的是早期 94-bit 布局；当前 control_unit.sv 已扩展为 130-bit（字段加宽到 16-bit）
- **独立验证**：Makefile 有 dump_cu target 单独回归译码逻辑；sva/ 内 CU-A01... 断言逐位绑定

三层验证：golden model → cocotb → SVA / formal

cocotb + iverilog

[test/](#)

- `to_fixed(val, frac_bits=8)`: Python 侧与 RTL 共用 Q8.8 约定
- `test_tpu.py` 跑完整 forward + backward 指令序列，逐点比对 NumPy 结果
- Makefile 每个模块一条 target: `dump_*.sv` 波形顶层 + gtkwave 查看

SVA assertions

[sva/](#)

- 12 组 `*_assertions.sv`, 覆盖 PE / systolic / VPU / UB / CU 各子模块
- `bind_all_assertions.sv` 用 `bind` 统一挂到 RTL, 不侵入设计代码
- 抓握手 `valid`、指针越界、`overflow sticky` 这类「不变量」

formal + golden

[docs/](#)



同一套 RTL 的三种去向：FPGA · FPGA · ASIC

mnist_demo/

DE1-SoC FPGA

- train_mnist.py 在 PC 训练，权重打包进 data/model/
- host/ 通过 JTAG 把模型与笔画数据送进板子，firmware/ 编排指令
- 板级跑 MNIST 手写数字推理，artifacts/ 留存输出

xor_demo/

最小闭环

- XOR: 2×2 阵列 + 完整 backward 的最小可训练案例
- sim/ 板级仿真脚本，十分钟读懂训练指令序列
- Quartus 工程与产物保留在 fpga/、artifacts/

tiny-tpu-hardened/

OpenLane ASIC

- rtl/: 加固后的可综合 Verilog 版本
- openlane/config.json + runs/: 物理设计输入与运行结果
- sim/ 聚焦回归，保证加固不改变行为

README · Future Steps ① 为这套 130-bit ISA 写编译器；② 把阵列扩展到 256×256 / 512×512——参数化骨架已经留在 SYSTOLIC_ARRAY_WIDTH 里

四个值得带走的设计决策

01 数据流即架构

Weight-stationary systolic + Unified Buffer 闭环：前向与反向是同一条环，差别只在 vpu_data_pathway 的 4 个 bit。

02 定点数是契约

Q8.8 贯穿全通路；舍入集中在 fxp_zoom 一处；sticky overflow 把「静默的错误」变成可观测信号。

03 指令即控制面板

130-bit 宽指令位切片直连控制线，无取指、无状态机——把复杂度从硬件推给编译器，而编译器正是 Future Steps 的第一条。

04 验证是交付物的一部分

NumPy golden → cocotb → SVA bind → formal 四层；断言矩阵与覆盖率报告跟代码一起进版本库。

3,162 行 SystemVerilog，一台能训练的完整 TPU。 *From MAC to gradient descent — every line is readable.*

总结：文件树 ↔ 运行时模块，一色一子系统

