SIMD Accelerator for Matrix Multiplication

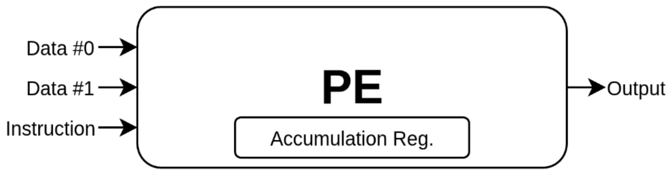
Luke Qiao, Aarav Wattal

Overview of Modules

Here, we outline the function of each module in the matrix multiplication accelerator.

Processing Element (PE)

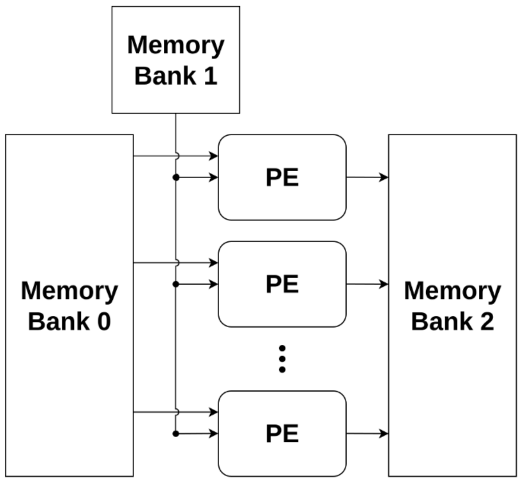
The **processing_element** module is the atomic compute unit of the SIMD accelerator, and is responsible for performing arithmetic operations under a shared instruction stream while operating on independent data.



Each PE receives two 32-bit input operands and an instruction composed of an Opcode, Mode, and Value field. The Mode selects the data bitwidth, 32-bit, 16-bit, or 8-bit, causing the PE to internally partition its accumulation register into multiple virtual registers for independent accumulation across sub-word lanes. The Opcode determines one of six behaviors: MAC (multiply-accumulate), PASS (load data0 into accumulation), CLR (zero internal state), NOP, OUT (truncate and write accumulation contents), or RND (right shift of accumulation by Value).

Main Buffer

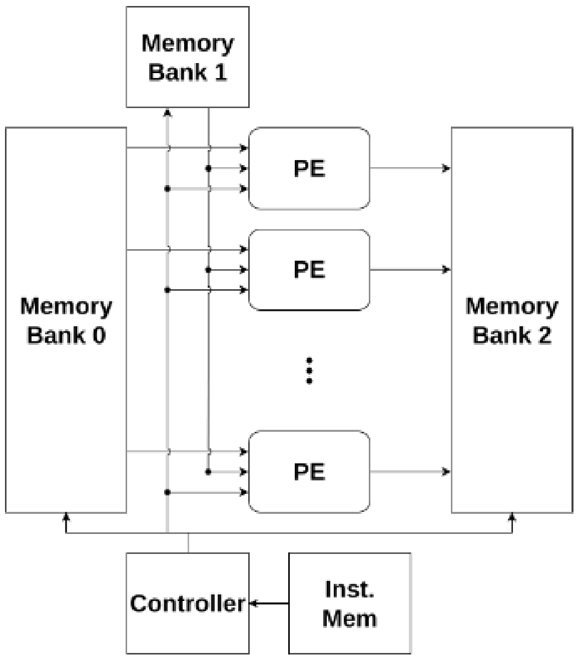
The **buffer** module is responsible for the memory access by interfacing three memory banks, MEM0, MEM1, and MEM2, to provide operands to the PE array and store results.



It receives a memory instruction containing Opcode, Mode, MemAOffset, and MemBOffset, using these fields to drive address decoding and data packing behavior. Under the READ opcode, MEM0 always returns a full 32-bit word, whereas MEM1 returns a replicated 8-bit or 16-bit sub-field selected via MemBOffset when operating in 16-bit or 8-bit modes; this selective replication ensures that each lane of the PE receives identical broadcasted B-operands for vector dot product semantics. Under WRITE, the Main Buffer stores the concatenated output from all PEs into MEM2 at MemAOffset. The memory subsystem enables the accelerator to store PE instructions and the matrix and vector data, which is critical for functionality.

Controller

The **controller** module coordinates all accelerator operations by issuing instructions to both the Main Buffer and the PE array, enforcing the single instruction, multiple data (SIMD) execution model.



Each controller instruction contains MemInstruction, PEInstruction, Count, MemAIncrement, and MemBIncrement. On execution, the controller forwards the embedded MemInstruction and PEInstruction to the Main Buffer and all PEs respectively, repeating this dispatch for Count + 1 cycles. During each repetition, it increments the MemAOffset and MemBOffset fields by the specified increments, enabling auto-incrementing memory access patterns tightly matched to row-by-column matrix multiplication traversal. The controller therefore handles loop unrolling, broadcast synchronization, and consistent memory/compute sequencing across all PEs, ensuring correct SIMD behavior without requiring distributed control logic inside individual processing elements.

Top-Level Accelerator

The top-level **accelerator** module integrates the Controller, Main Buffer, and Processing Element array into a unified matrix multiplication accelerator. It loads machine instructions generated by the assembler/compiler stored in memory, steps through them sequentially, and for each instruction issues the appropriate control signals to the Controller, which in turn orchestrates the Main Buffer and PE operations. The top module instantiates a configurable number of PEs (determined by the design configuration, 16 by default), wires their inputs to MEM0 and MEM1 outputs, and wires their outputs back into MEM2 through the Main Buffer's write path. The accelerator enforces correct ordering of effects across memory reads, PE arithmetic, accumulation register updates, truncation/rounding logic, and memory writes, effectively modeling the microarchitectural timing of the SIMD pipeline. It must also maintain global state, including the program counter, instruction memory, and internal module states, to ensure deterministic execution of matrix multiplication kernels generated by the provided compiler. Overall, the top-level module is responsible for simulating the entire instruction-driven dataflow of the accelerator, ensuring correctness of instruction sequencing, and the collective behavior of all PEs under the SIMD execution model.

Hardware Optimization

Note we optimize the figure-of-merit (FoM) given by:

$$\mathrm{FoM} = (\mathrm{Time_{int8}} + \mathrm{Time_{int16}} + \mathrm{Time_{int32}})^{2.3}(\mathrm{Area}\ \mu\mathrm{m}^2)(\mathrm{Power\ mW}) \cdot 10^{-15}$$

We use Synopsis Design Compiler for synthesis.

Hardware Reuse

Description: In the original processing element (PE) implementation, we instantiated separate multipliers for each data mode (INT8, INT16, INT32). A naive design would use four independent 8-bit multipliers, two 16-bit multipliers, and one 32-bit multiplier. Since only one mode is active at any time, most of this hardware would remain idle, leading to significant waste.

Our optimized design instead reuses hardware by implementing four shared 32-bit multipliers (mult0 through mult3). These are dynamically configured depending on the operating mode. In INT32 mode, only mult0 is used for the full 32-bit multiplication. In INT16 mode, mult0 and mult1 perform two 16-bit multiplications with sign extension. In INT8 mode, all four multipliers operate in parallel to handle four 8-bit multiplications.

Operands are sign-extended to 32 bits through a combinational block before multiplication. For example, smaller values are sign-extended before being fed into the shared multipliers.

```
mult0_a = {{24{vec8_0[7]}}, vec8_0}; // sign-extend to 32-bit
mult0_b = {{24{mat8_0[7]}}, mat8_0};
```

FoM Change: This optimization reduced the average area of each PE from about 11,500.9 µm² to 10,202.4 µm², an 11.29% area reduction. This lowered the total chip area and power, improving the overall FoM by approximately 13.56%.

Reflection: This demonstrated the importance of allocating hardware based on actual usage instead of dedicating independent resources for every possible mode. Removing unnecessary multipliers dramatically reduced area without hurting functionality.

Register Re-Timing

Description: We enabled automatic register retiming in Design Compiler using the "-retime" flag to the compile_ultra command in the synthesis script.

```
compile_ultra -retime -gate_clock -timing_high_effort_script
```

Retiming allows the synthesis tool to move registers across combinational logic to balance the delays across pipeline stages. This is particularly beneficial when one stage has significantly deeper logic than others.

FoM Change: Register retiming helped us achieve timing closure at a 2.0 ns clock period (500 MHz). The synthesis logs confirmed that the tool moved registers to equalize path delays. This optimization improved the overall FoM by about 5.28%.

Reflection: Retiming is powerful because the tool has a global view of the timing graph, allowing it to rebalance pipelines more effectively than manual tuning. However, the tool could not fully retime the accumulator feedback loop because of structural constraints. This motivated us to manually pipeline the MAC path. Combining automatic retiming with manual pipelining resolved the timing bottlenecks.

Timing Optimization

Description: We manually pipelined the processing element to break up the long combinational path through the multipliers and accumulators. The critical path originally ran from inputs, through the multiplier, through the accumulation adder, and into the accumulator register. This path was too long for our target clock period.

We inserted pipeline registers between the multiplier and accumulator. After this change, the first stage performs the multiplication and registers its results, and the second stage performs accumulation. Instruction decode signals (such as MAC enable and mode bits) were also pipelined so they remained synchronized with the delayed multiplier outputs. Not all instructions need the multiplier pipeline. Simple instructions such as PASS, CLR, RND, and OUT have short critical paths, so we added bypass paths that skip the multiplier pipeline to avoid unnecessary latency and wasted clock cycles.

The final synthesized design met timing with zero slack at a 2.0 ns clock, with a worst-case path of 1.96 ns. We see that there are no timing errors, as shown in the timing report in compile.log.

```
...
Critical Path Slack:          0.00
Critical Path Clk Period:     2.00
Total Negative Slack:         0.00
No. of Violating Paths:       0.00
Worst Hold Violation:         0.00
Total Hold Violation:         0.00
No. of Hold Violations:       0.00
```

FoM Change: This optimization was essential for meeting timing. It enabled a 31.55% improvement in maximum frequency compared to the unpipelined version.

Reflection: Manual pipelining required identifying which instruction paths were actually timing-critical. MAC operations dominated the critical path, while simpler instructions did not benefit from additional pipelining. Selectively pipelining only the MAC path allowed us to meet timing without penalizing other operations. The synthesis timing report guided us in locating and breaking the true critical paths.

Clock Gating

Description: We enabled automatic clock gating in Design Compiler by setting an appropriate clock gating style and using the "-gate_clock" flag.

```
set_clock_gating_style -sequential_cell latch \
    -positive_edge_logic {integrated} \
    -negative_edge_logic {or} \
    -control_point before \
    -control_signal scan_enable
set_clock_gating_check -setup 0.5 -hold 0.1
compile_ultra -retime -gate_clock -timing_high_effort_script
```

Clock gating inserts enable logic before registers, preventing unnecessary clock toggling when registers do not need updating. The synthesis logs confirmed that clock gating was applied to all 16 PEs, and integrated clock-gating cells were inserted on the accumulator registers.

```
Information: Performing clock-gating on design \
    processing_element_0. (PWR-730)
Information: Performing clock-gating on design \
    processing_element_1. (PWR-730)
...
```

FoM Change: Clock gating reduced power consumption from about 32.69 mW to 8.77 mW. This produced a major improvement in overall FoM of roughly 73.17%. Power here is defined as total switching plus leakage power.

Reflection: Clock gating is especially effective in SIMD designs where not all processing elements are active at all times. The accumulator registers are only used during certain operations, so preventing unnecessary switching during idle periods significantly reduces power. The synthesis tool inferred enable signals from the RTL and applied gating automatically.

Design Sizing

Description: We set the design to use 16 processing elements. This was chosen as a balance between high parallelism and area constraints. According to the synthesis report, the total cell area is about 164,986 µm², and the PEs account for around 98% of this. Each PE contributes about 10,202.4 µm². With 16 PEs, we remain well under the 1,000,000 µm² project limit while achieving good performance.

FoM: Because the FoM formula weights execution time with an exponent of 2.3, reducing runtime has an outsized impact relative to area or power. With 16 PEs at 500 MHz, our test workloads complete in roughly 33.01 µs (INT8), 16.62 µs (INT16), and 8.43 µs (INT32).

Reflection: Choosing the number of PEs required understanding how the FoM scales with time, area, and power. Adding more PEs reduces time but increases area, so the benefit does not scale linearly. We found that 16 PEs offered strong parallelism without excessive area growth. Future designs might explore heterogeneous PEs or dynamic PE allocation.

Final FoM

Our final FoM is

$$\mathrm{FoM} = (33.01\,\mu\mathrm{s} + 16.62\,\mu\mathrm{s} + 8.43\,\mu\mathrm{s})^{2.3}(164985.9\,\mu\mathrm{m}^2)(8.77\,\mathrm{mW}) \cdot 10^{-15} \approx 131.03$$

which is a substantial improvement from the initial FoM of 871.35.