Kepler-Style SIMT GPU Core

Architecture Specification & Datasheet

Specification Overview

Educational GPGPU Processor
5-Stage Pipelined SIMT Architecture
Dual-Issue Superscalar Execution
Hardware 3D Graphics Orchestration
768 Concurrent Threads (24 Warps × 32 Threads)

Document Version: 1.0
January 22, 2026

Contents

1 Executive Summary

The **Kepler-Style SIMT Core** is an educational, behavioral model of a General Purpose GPU (GPGPU) processor. Designed for learning and architectural exploration, it implements a 32-thread Single Instruction, Multiple Threads (SIMT) architecture compliant with modern GPU execution models.

1.1 Key Features

- **Architecture**: 5-Stage Pipelined SIMT Core (IF, ID, OC, EX, WB)
- **Parallelism**: 32 Threads per Warp, dual-issue capability (up to 2 instructions/cycle)
- **Multithreading**: Fine-Grained Multithreading (FGMT) with 24 warps (768 threads) and zero-overhead context switching
- **Memory Model**:
  - Shared Memory: 16KB On-Chip Scratchpad (32 banks) with bank-conflict replay
  - Global Memory: Coalesced LSU with **Multi-Line Split Support**
  - MSHR: 64-entry out-of-order tracking per warp
- **Synchronization**: Hardware Barrier (BAR) with Epoch consistency
- **Control Flow**: Hardware Divergence Stack (SSY/JOIN) and Function Call Stack (CALL/RET)
- **Graphics**: Hardware-accelerated 3D wireframe rendering with Perspective Projection

1.2 Main Implementation

The core logic is implemented in `streaming_multiprocessor.sv`, which contains the complete 5-stage pipeline, warp scheduler, scoreboard, and execution units.

2 Microarchitecture Specification

2.1 Pipeline Overview

The core implements an in-order, dual-issue pipeline with out-of-order memory completion.

Stage	Function
IF	Instruction Fetch - Warp scheduler selects ready warp, fetches 2 consecutive instructions
ID	Instruction Decode & Issue - Dual-issue logic checks dependencies and structural hazards
OC	Operand Collector - Gathers operands from banked register file, handles conflicts
EX	Execute & Memory - ALU, FPU, SFU operations; LSU calculates addresses and performs cache access
WB	Writeback - Arbitrates results from ALU, FPU, Memory; clears scoreboard

Table 1: Pipeline Stage Descriptions

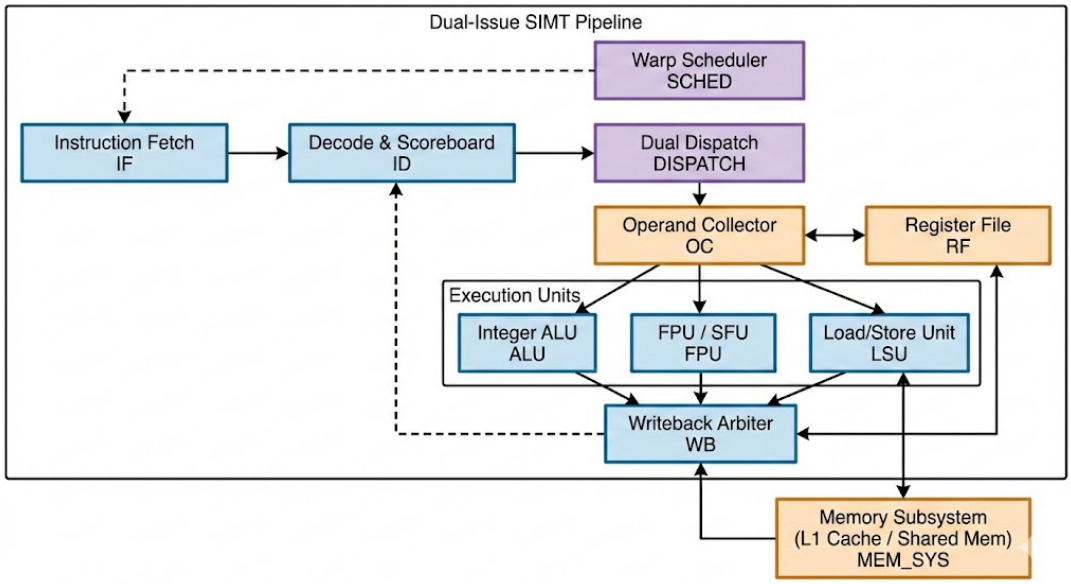


Figure 1: High-Level Pipeline Overview

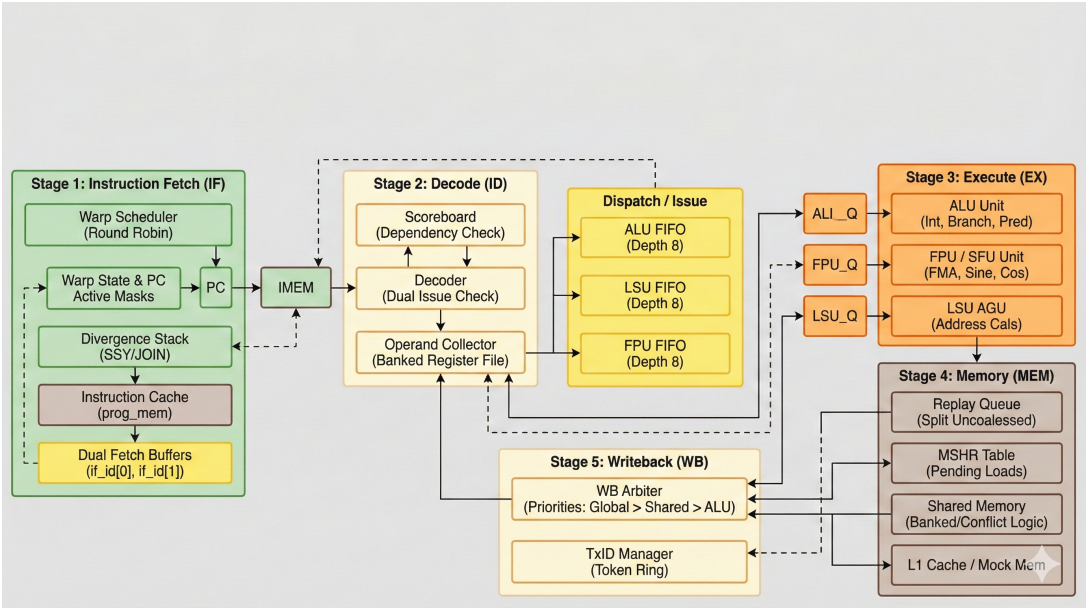


Figure 2: Detailed Dual-Issue SIMT Microarchitecture

2.2 Thread Hierarchy & Scheduling

Warp Organization

- **32 Threads per Warp**: Executed in lockstep (SIMT)
- **24 Warps per Core**: Supporting up to 768 concurrent threads resident on the SM
- **Warp ID**: 5-bit identifier (0-23)

2.2.1 Multithreading Model

- **Fine-Grained Multithreading (FGMT)**: The core can switch between warps on a cycle-by-cycle basis to hide latency
- **Latency Hiding**: When a warp stalls (scoreboard dependency, memory wait, barrier), the scheduler immediately switches to another ready warp

2.2.2 Scheduling & Switching Rule

- **Greedy Scheduling**: The scheduler stays with the same warp as long as it has eligible instructions
- **When do Warps Switch?**:
  1. The current warp stalls (RAW dependency, memory operation pending, barrier wait)
  2. The current warp completes (hits EXIT or diverges to inactive state)
- **Two-Loop Search Mechanism**:
  - Outer Loop: Iterates through all 24 warp slots starting from `rr_ptr`
  - Inner Check: Checks eligibility (state = READY, no scoreboard conflicts, OC has space)
  - First-Match: Selects the first eligible warp found and breaks
  - Pointer Update: After selection, `rr_ptr` advances to `(selected_warp + 1) % 24`
- **Context Switching Overhead**: Zero Cycles - all warp state is hardware-resident

2.3 Dual-Issue Compatibility

Instruction A	Can Pair With	Cannot Pair With	Reason
ALU	FPU, LSU, SFU	ALU, CTRL	Same functional unit conflict
FPU	ALU, LSU, SFU	FPU	Same functional unit conflict
LSU	ALU, FPU, SFU	LSU	Only 1 memory port per warp
SFU	ALU, FPU, LSU	SFU	Same functional unit conflict
CTRL	None	All	Control flow must execute alone

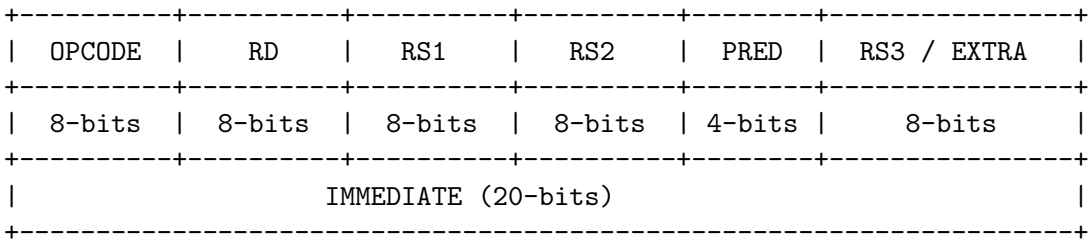
Table 2: Dual-Issue Compatibility Matrix

Additional Constraints

- **RAW Hazard**: Instruction B cannot read a register that A writes
- **WAW Hazard**: Both instructions cannot write to the same destination register
- **Control Flow**: Branches, barriers, synchronization, and function calls always issue alone

3 Instruction Set Architecture

3.1 Instruction Encoding Format (64-bit)



- **OPCODE**: Specifies the operation (e.g., ADD, LDR, BRA)
- **RD**: Destination Register Index (R0-R63)
- **RS1 / RS2**: Source Register Indices
- **PRED**: Predicate Register Index (P0-P7) and Condition Flags
- **RS3 / EXTRA**: Third Source Register or Branch Target Offset
- **IMMEDIATE**: 20-bit Signed Immediate / Offset

3.2 Opcode Map (Selected Instructions)

Opcode	Mnemonic	Description
Integer Arithmetic & Logic		
0x00	NOP	No Operation
0x01	ADD	Integer Addition
0x02	SUB	Integer Subtraction
0x03	MUL	Integer Multiplication
0x10	AND	Bitwise AND
0x11	OR	Bitwise OR
0x12	XOR	Bitwise XOR
Memory Operations		
0x20	LDR	Load from Global Memory
0x21	STR	Store to Global Memory
0x22	LDS	Load from Shared Memory
0x23	STS	Store to Shared Memory
Control Flow		
0x24	BRA	Unconditional Branch
0x25	BEQ	Branch if Equal
0x26	BNE	Branch if Not Equal
0x27	CALL	Function Call
0x28	RET	Return from Function
0x29	SSY	Set Synchronization Point
0x2A	JOIN	Re-converge Divergent Paths
0x2B	BAR	Barrier Synchronization
0x2C	EXIT	Terminate Warp Execution

Table 3: Instruction Set Architecture (Partial)

4 Subsystem Deep Dive

4.1 Operand Collector (OC)

The Operand Collector decouples instruction fetching from operand reading, resolving register file bank conflicts.

Key Mechanism

1. Instructions are allocated to a Collector Unit (CU)
2. The CU requests operands from the Bank Arbiter
3. The Arbiter grants access based on bank availability (Bank = RegID % 4)
4. Once all operands are collected, the CU dispatches to Execution Units

4.2 Divergence Stack (SSY/JOIN)

To handle SIMT divergence on control flow, the core maintains a per-warp Divergence Stack.

Divergence & Serialization:

1. Pushes current Active Mask, PC, and Token onto stack via SSY
2. Updates Active Mask to only enable threads taking the branch
3. Serializes execution: "then" path executes first, then "else" path
4. After all paths complete, JOIN pops the stack and restores full Active Mask

Performance Impact

Divergent paths are executed **serially** (one after another), not in parallel, which can reduce effective throughput when warps diverge frequently.

The core implements a robust memory hierarchy with a non-blocking Coalesced Load/Store Unit (LSU).

LSU Split Handling: The LSU automatically detects when a warp's memory accesses span multiple 128-byte cache lines. It utilizes a hardware-managed **Replay Queue** to serialize these into multiple sequential requests to the memory system. This process is transparent to the warp scheduler, allowing uncoalesced requests to progress without software intervention.

Component	Description
MSHR Table	64-entry table per warp tracking pending memory operations
Transaction ID	16-bit ID: [5:0] Slot ID, [9:6] Warp ID, [15:10] SM ID
FIFO Management	Free IDs managed via per-warp FIFO (pop on issue, push on response)
Scoreboard	Prevents RAW hazards by blocking dependent instructions until response arrives

Table 4: Memory Subsystem Components

Data Hazard Prevention: Even with out-of-order completion, RAW hazards are prevented through the scoreboard. When a load issues, the destination register is immediately scoreboarded. Dependent instructions stall at ID stage until the memory response clears the scoreboard.

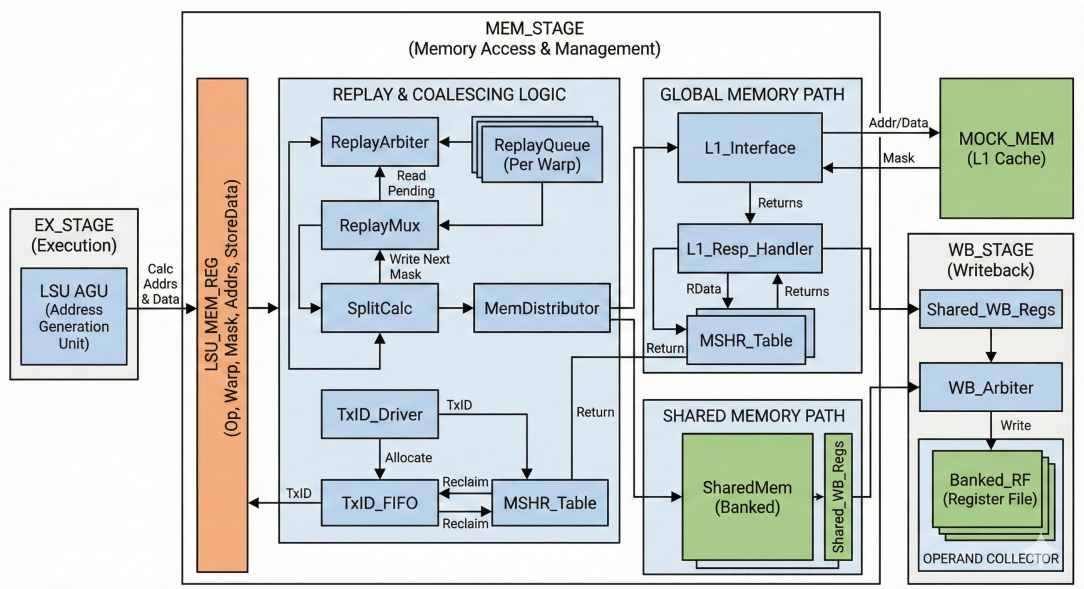


Figure 3: LSU and Memory Subsystem Architecture

4.3 Barrier Synchronization (Epoch Consistency)

The hardware barrier (BAR) implements epoch consistency to prevent fast-warp race conditions.

Epoch Mechanism

- **Problem**: Fast warp could re-enter same barrier before slow warp exits
- **Solution**: Global `barrier_epoch` bit toggles on each barrier resolution
- **Guarantee**: Warps only contribute if their local epoch matches global epoch

Each thread has 7 predicate registers (P0-P6), plus implicit P7 (always true).

Listing 1: Predicate Example

```
ISETP.GT P1, R_x, 0     // Set P1 = (x > 0)
@P1 MUL R_y, R_x, 2     // Only execute if P1 is true
```

Execution mask: `exec_mask = warp_active_mask & predicate_mask`

4.4 3D Graphics Orchestration

The core features dedicated support for 3D graphics vertex processing. By leveraging the SFU for rotation math and high-throughput integer arithmetic for perspective projection, the SM can orchestrate full 3D wireframe rendering.

- **Perspective Projection**: Hardware-accelerated depth scaling using `IDIV` for W-clipping and perspective divide.
- **Rotation Engine**: High-precision 1.15 fixed-point rotation matrices utilizing the SFU's `SIN` and `COS` units.
- **Vertex Shader Pipeline**: Full assembly-level implementation of vertex transformation directly writing to a global framebuffer.

5 Verification & Performance

The core includes a comprehensive verification suite focused on GPGPU and Graphics kernels.

5.1 Benchmark: 8×8 Tiled Matrix Multiplication

This benchmark verifies $C = A \times B$ for 8×8 matrices using shared memory tiling.

5.1.1 Why Tiling & Shared Memory?

Naive Implementation	**Tiled Implementation**
64 global memory accesses/thread	8 global memory accesses/thread
∼3000+ cycles	487 cycles
High latency (100-400 cycles)	Low latency (1-2 cycles for shared mem)

Table 5: Performance Comparison

Benefits:

1. **Data Reuse**: Load each tile once, reuse across all threads (∼8× reduction in global memory traffic)
2. **Low Latency**: Shared memory has ∼100× lower latency than global memory
3. **Bandwidth Efficiency**: Coalesced loads maximize memory bandwidth utilization

5.2 Performance Metrics

Metric	Value
Total Execution Cycles	487
Matrix Size	8×8
Threads per Warp	32
Warps Used	2
Shared Memory Usage	512 bytes

Table 6: Matrix Multiplication Performance

Verification: The simulation initializes Matrix A with linear values (1..64) and Matrix B as an Identity matrix. The expected result C is identical to A, which the testbench verifies successfully in 487 cycles.

```
Verify: Matrix A (Input)
  [    1    2    3    4    5    6    7    8 ]
  [    9   10   11   12   13   14   15   16 ]
  [   17   18   19   20   21   22   23   24 ]
  [   25   26   27   28   29   30   31   32 ]
  [   33   34   35   36   37   38   39   40 ]
  [   41   42   43   44   45   46   47   48 ]
  [   49   50   51   52   53   54   55   56 ]
  [   57   58   59   60   61   62   63   64 ]

Verify: Matrix B (Input)
  [    1    0    0    0    0    0    0    0 ]
  [    0    1    0    0    0    0    0    0 ]
  [    0    0    1    0    0    0    0    0 ]
  [    0    0    0    1    0    0    0    0 ]
  [    0    0    0    0    1    0    0    0 ]
  [    0    0    0    0    0    1    0    0 ]
  [    0    0    0    0    0    0    1    0 ]
  [    0    0    0    0    0    0    0    1 ]

Verify: Matrix C (Result)
  [    1    2    3    4    5    6    7    8 ]
  [    9   10   11   12   13   14   15   16 ]
  [   17   18   19   20   21   22   23   24 ]
  [   25   26   27   28   29   30   31   32 ]
  [   33   34   35   36   37   38   39   40 ]
  [   41   42   43   44   45   46   47   48 ]
  [   49   50   51   52   53   54   55   56 ]
  [   57   58   59   60   61   62   63   64 ]
====================================================

TEST PASSED!
Total Cycles: 487
```

5.3 Benchmark: 3D Graphics – Perspective Cube

A dedicated graphics kernel (`test_perspective_cube.sv`) validates the core's ability to orchestrate 3D vertex processing.

- **Shader Logic**: Performs 3D rotations and perspective transformation ($x' = x \cdot f/z$).
- **Hardware Utilization**: Stresses the SFU (trigonometry), IDIV (projection), and the Replay Queue (framebuffer stores).
- **Verification**: Bit-accurate comparison of projected vertex coordinates against a golden Python reference model.

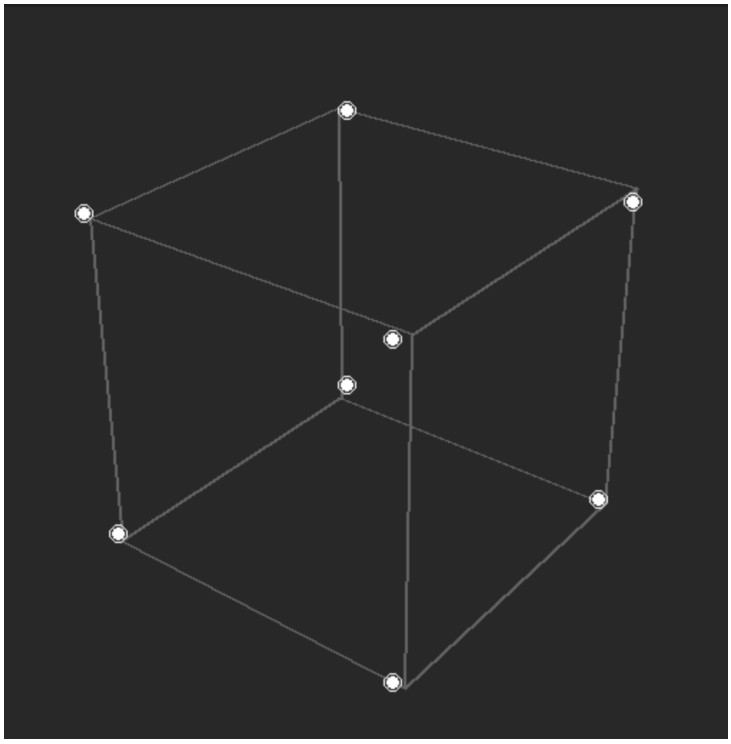


Figure 4: 3D Wireframe Rendering Output

5.4 Benchmark: Parallel Vertex Processing

A parallelized version of the vertex shader (`test_parallel_cube.sv`) was developed to utilize SIMT execution (8 threads).

Mode	Total Cycles	Speedup
Sequential (1 Thread)	99,312	1.0x
Parallel (8 Threads)	34,368	**2.9x**

Table 7: Vertex Processing Performance

Analysis: The parallel implementation achieves a nearly 3x speedup. The theoretical 8x speedup is constrained by the lack of hardware atomic operations, necessitating a serialized framebuffer write loop (software ROP) to prevent race conditions.

5.5 Regression Suite

The core is validated against an 11-test regression suite covering all architectural features:

Test Name	Feature Validated
test_alu_ops	Basic integer arithmetic and logic
test_app_matmul	8x8 Tiled Matrix Multiplication
test_control_flow	Nested branches and divergence stack
test_fpu_sfu_ops	Floating point and transcendental units
test_function_call	CALL/RET hardware stack
test_lsu_split	LSU memory coalescing and multi-line splits
test_memory_system	MSHR and transaction tracking
test_perspective_cube	3D Rendering with Perspective Projection
test_pipeline_issue	Dual-issue structural and data hazards
test_rotated_cube	Orthographic 3D rotation shader
test_wireframe_cube	Basic wireframe rendering demo

Table 8: SystemVerilog Regression Suite

6 Limitations & Future Work

6.1 Current Limitations

Rasterization Stage

The current hardware supports point-based vertex rendering and wireframe logic. Full-triangle rasterization with barycentric interpolation is a future milestone for the dedicated rasterization unit.

6.2 Future Enhancements

- L1/L2 cache hierarchy integration with genuine set-associative logic
- Multi-SM scaling with Network-on-Chip (NoC) interconnect
- Hardware Rasterizer and Texture Mapping Unit (TMU) integration
- Performance counters and GPGPU profiling support

7 Conclusion

The Kepler-Style SIMT Core provides a comprehensive, educational implementation of modern GPU architecture principles. With its dual-issue pipeline, hardware divergence handling, and sophisticated memory subsystem, it serves as an excellent platform for learning GPU microarchitecture and SIMT execution models.

For more information, refer to the RTL implementation in:
`RTL/Core/streaming_multiprocessor.sv`