Kepler-Style SIMT GPU Core

Architecture Specification & Datasheet

Specification Overview

Educational GPGPU Processor
6-Stage Pipelined SIMT Architecture
Dual-Issue Superscalar Execution
768 Concurrent Threads (24 Warps × 32 Threads)

Document Version: 1.0
January 19, 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**: 6-Stage Pipelined SIMT Core (IF, ID, OC, EX, MEM, 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)
  - Global Memory: Coalesced Load/Store Unit (LSU)
- **Synchronization**: Hardware Barrier (BAR) with Epoch consistency
- **Control Flow**: Hardware Divergence Stack (SSY/JOIN) for nested branches
- **Predication**: 7 predicate registers per thread for conditional execution

1.2 Main Implementation

The core logic is implemented in `streaming_multiprocessor.sv`, which contains the complete 6-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 - ALU, FPU, SFU, Branch Unit
MEM	Memory - LSU handles global/shared memory operations
WB	Writeback - Arbitrates results from ALU, FPU, Memory; clears scoreboard

Table 1: Pipeline Stage Descriptions

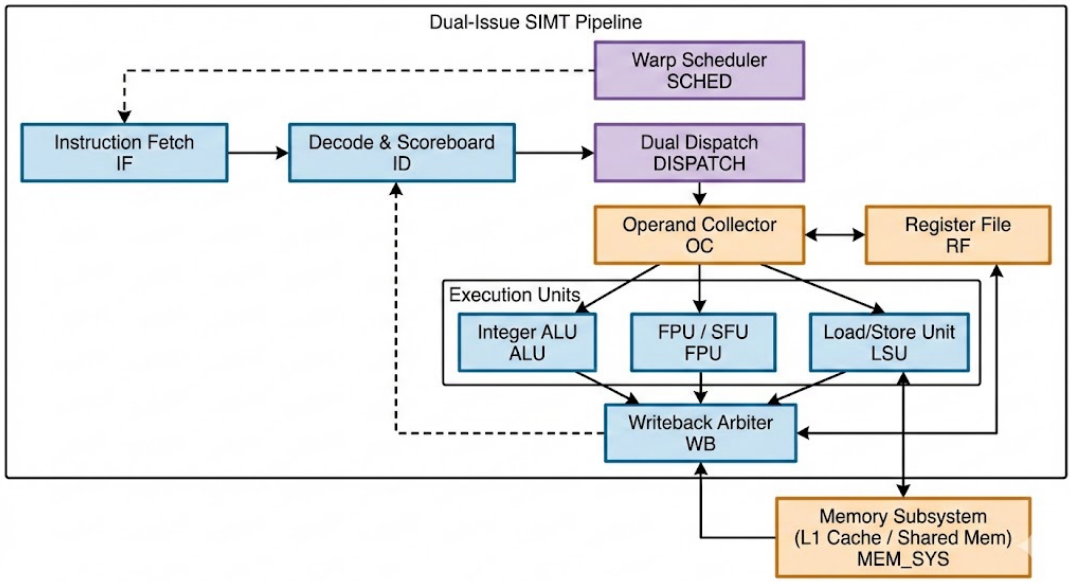


Figure 1: Dual-Issue SIMT Pipeline Architecture

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

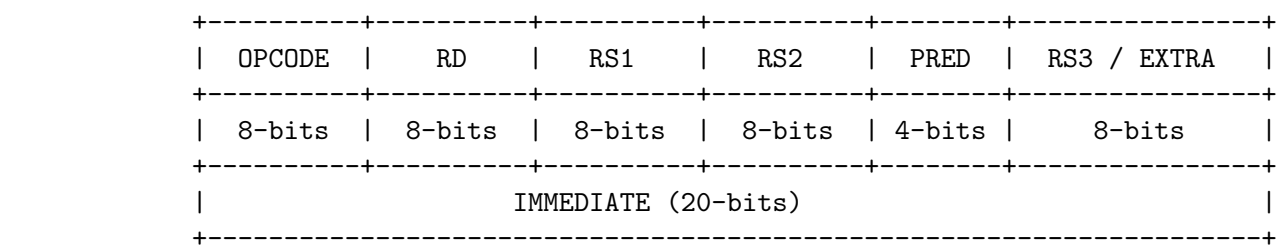
Table 2: Dual-Issue Compatibility Matrix

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)

4 Subsystem Deep Dive

4.1 Operand Collector (OC)

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

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)

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.

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

4.3 Memory Subsystem (MSHR & Transaction Tracking)

The core implements out-of-order memory completion to hide long memory latencies.

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.

4.4 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

4.5 Predicated Execution

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

Listing 1: Predicate Example

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

Execution mask: `exec_mask = warp_active_mask & predicate_mask`

5 Verification & Performance

5.1 Benchmark: 8×8 Tiled Matrix Multiplication

The core includes a testbench that verifies a Tiled Matrix Multiplication algorithm performing $C = A \times B$ for 8×8 matrices.

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
```

6 Limitations & Future Work

6.1 Current Limitations

Memory Coalescing

The current LSU implementation supports only one cache line request per warp per instruction. Uncoalesced accesses (spanning multiple cache lines) result in undefined behavior. A production implementation requires a SPLIT/REPLAY mechanism.

6.2 Future Enhancements

- LSU Splitter FSM for uncoalesced memory access handling
- L1/L2 cache hierarchy integration
- Multi-SM scaling with interconnect
- Performance counters and 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`