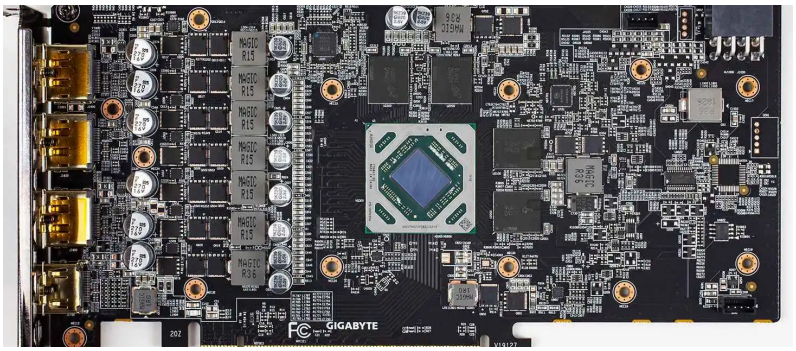
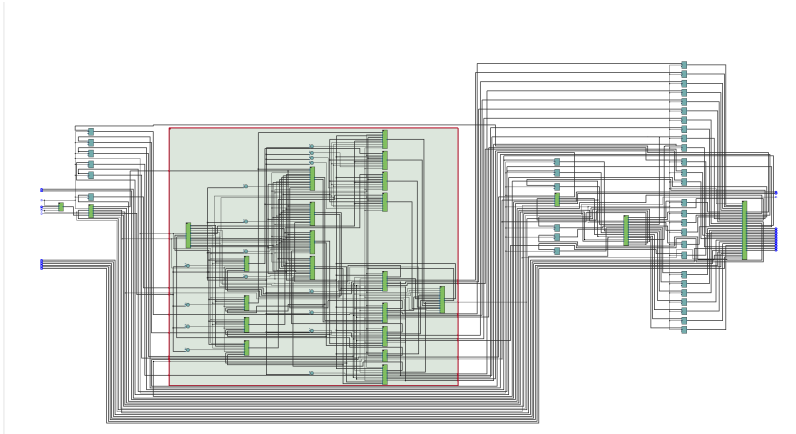
Understanding GPU Architecture and Mini-GPU Implementation

Harsha Sai Srinivas Pamarthi

Seasons of Code (2025)
Web and Coding Club, IIT Bombay

May '25 – July '25





Abstract

This comprehensive report demystifies Graphics Processing Unit (GPU) architecture through both conceptual explanations and technical analysis of *tiny-gpu* — a minimal educational GPU implementation in System Verilog. Beginning with fundamental concepts accessible to non-specialists, we progressively explore GPU internals using analogies, diagrams, and practical examples. The document features a complete breakdown of tiny-gpu's architecture, instruction set, and execution model, demonstrating how parallel computation principles are implemented in hardware. Designed as both an introductory guide and technical reference, this work bridges the gap between theoretical concepts and practical hardware implementation.

Contents

1 Introduction to GPU Architecture

1.1 The Evolution of Parallel Processing

Graphics Processing Units (GPUs) represent one of the most significant innovations in computing history. Originally designed to accelerate 3D graphics rendering, these specialized processors have evolved into general-purpose parallel computing engines. The key to their power lies in a fundamental architectural difference from Central Processing Units (CPUs):

- **CPU Philosophy:** Optimized for *sequential task execution*
  - Fewer complex cores (typically 2-64)
  - Sophisticated control logic for instruction scheduling
  - Large cache hierarchies minimizing memory latency
  - Excels at tasks with complex decision-making
- **GPU Philosophy:** Optimized for *parallel data processing*
  - Thousands of simplified processing cores
  - Minimal control logic per core
  - High-bandwidth memory interfaces
  - Excels at repetitive, data-parallel operations

Historical Context: The GPU revolution began in 1999 with NVIDIA's GeForce 256, marketed as "the world's first GPU." This chip contained approximately 22 million transistors and could process 10 million polygons per second. Modern GPUs like NVIDIA's H100 contain 80 billion transistors and deliver nearly 2,000 TFLOPS of compute performance.

1.2 Why Parallel Processing Matters

The demand for parallel processing stems from fundamental physical constraints known as Dennard scaling and Moore's Law. As transistor counts continued doubling approximately every two years (Moore's Law), clock speeds plateaued due to power density limitations (Dennard scaling breakdown). This necessitated a shift from faster sequential processing to parallel computation:

$$\text{Total Work} = \text{Work per Core} \times \text{Number of Cores}$$

The Parallelism Advantage: Consider processing a 4K UHD image (3840×2160 pixels = 8.3 million pixels):

- **Sequential Approach (1 core):** 8.3 million operations × 10 ns/op = 83 ms
- **Parallel Approach (10,000 cores):** 8.3 million ops / 10,000 cores × 10 ns/op = 8.3 μs

This 10,000× speedup demonstrates why GPUs have become essential for modern computing workloads.

1.3 GPU Application Domains

Modern GPUs accelerate computation across diverse fields:

Domain	GPU Application
Computer Graphics	Real-time rendering for games/virtual reality, ray tracing, anti-aliasing
Scientific Computing	Molecular dynamics, fluid dynamics, quantum chemistry simulations
Artificial Intelligence	Training deep neural networks, natural language processing
Medical Imaging	MRI reconstruction, CT scan processing, protein folding
Financial Modeling	Risk analysis, algorithmic trading, Monte Carlo simulations
Cryptocurrency	Proof-of-work hashing, blockchain validation
Autonomous Vehicles	Sensor fusion, path planning, real-time object detection
Climate Science	Atmospheric modeling, weather prediction, climate simulations

Table 1: Expanded GPU Application Domains

2 Core GPU Concepts

2.1 The SIMD Execution Model

Graphics Processing Units (GPUs) fundamentally operate on the principle of Single Instruction, Multiple Data (SIMD) processing. This parallel computing architecture executes a single instruction simultaneously across multiple data elements, mathematically represented as:

$$\text{Instruction}_k \rightarrow \text{Data}_1, \text{Data}_2, \ldots, \text{Data}_n \tag{1}$$

Where:

- Instruction_k is a single operation (e.g., ADD, MUL)
- Data_1 to Data_n are distinct data elements processed in parallel
- n represents the number of processing elements (cores/threads)

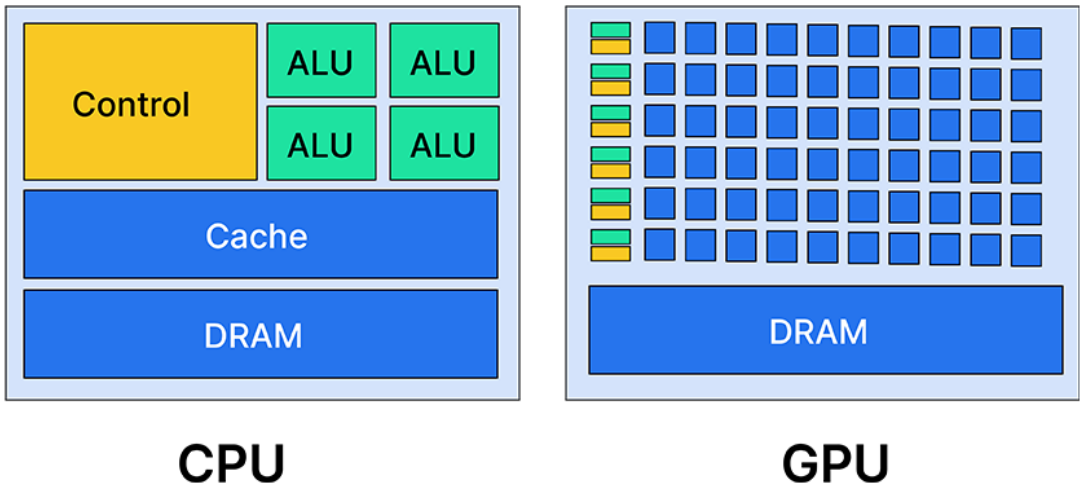


Figure 1: CPU vs GPU

2.1.1 Detailed Analogy: Classroom Instruction

Consider teaching mathematics to a classroom:

- **CPU Approach (SISD):**
  - A tutor works with each student individually
  - Each student receives personalized instruction: "Alice, calculate 5+3. Bob, calculate 7+2..."
  - Efficient for different tasks but slow for identical operations
  - *Hardware Equivalent:* Sequential execution on single core
- **GPU Approach (SIMD):**
  - Professor announces: "Everyone, add your two numbers!"
  - All students perform addition simultaneously on their unique data
  - Alice computes 5+3, Bob computes 7+2, etc., at the same time
  - *Hardware Equivalent:* Parallel execution across multiple cores

2.1.2 SIMD vs. SIMT: Evolution of Parallel Execution

Modern GPUs implement a refined model called Single Instruction, Multiple Threads (SIMT):

Characteristic	**SIMD**	**SIMT**
Execution Unit	Vector processor	Scalar processors
Control Flow	Lockstep execution	Divergent threads allowed
Programming Model	Explicit vectorization	Implicit parallelism
Data Access	Uniform memory access	Per-thread memory access
Flexibility	Rigid data alignment	Independent memory access
Hardware Cost	Complex vector units	Multiple simple cores

Table 2: SIMD vs. SIMT Architectural Differences

2.1.3 Hardware Implementation

The SIMT model is implemented through the following hardware mechanisms:

1. **Warp Formation:**
   - Threads grouped into warps (typically 32 threads)
   - All threads in warp share instruction fetch/decode
   - Example: NVIDIA warp, AMD wavefront
2. **Instruction Dispatch:**
   - Single instruction issued per warp cycle
   - All active threads execute same instruction
   - Warp scheduler manages multiple warps
3. **Data Distribution:**
   - Each thread processes different data element
   - Memory access through unique thread IDs:

$$\text{Address}_i = \text{Base} + \text{Stride} \times \text{ThreadID}$$

2.2 Key GPU Terminology

Term	Definition
Kernel	A program designed to run on GPU cores
Thread	Smallest execution unit with own registers/state
Warp/ Wavefront	Group of threads (32-64) executing together
Block/ Workgroup	Collection of threads sharing resources
Grid	Collection of blocks executing same kernel
Register File	Fast memory private to each thread
Shared Memory	Fast memory shared within a block
Global Memory	High-latency main GPU memory

Table 3: GPU Terminology Reference

2.3 Memory Hierarchy

GPUs employ sophisticated memory systems to overcome the "memory wall" - the performance gap between processor speed and memory access times:

- **Registers (1 cycle access):** Fastest storage, private to each thread
- **Shared Memory (1-10 cycles):** On-chip memory shared within thread block
- **L1/L2 Cache (10-100 cycles):** Hardware-managed cached data
- **Global Memory (200-400 cycles):** High-latency DRAM storage

3 Tiny-GPU Architecture

3.1 System-Level Architecture

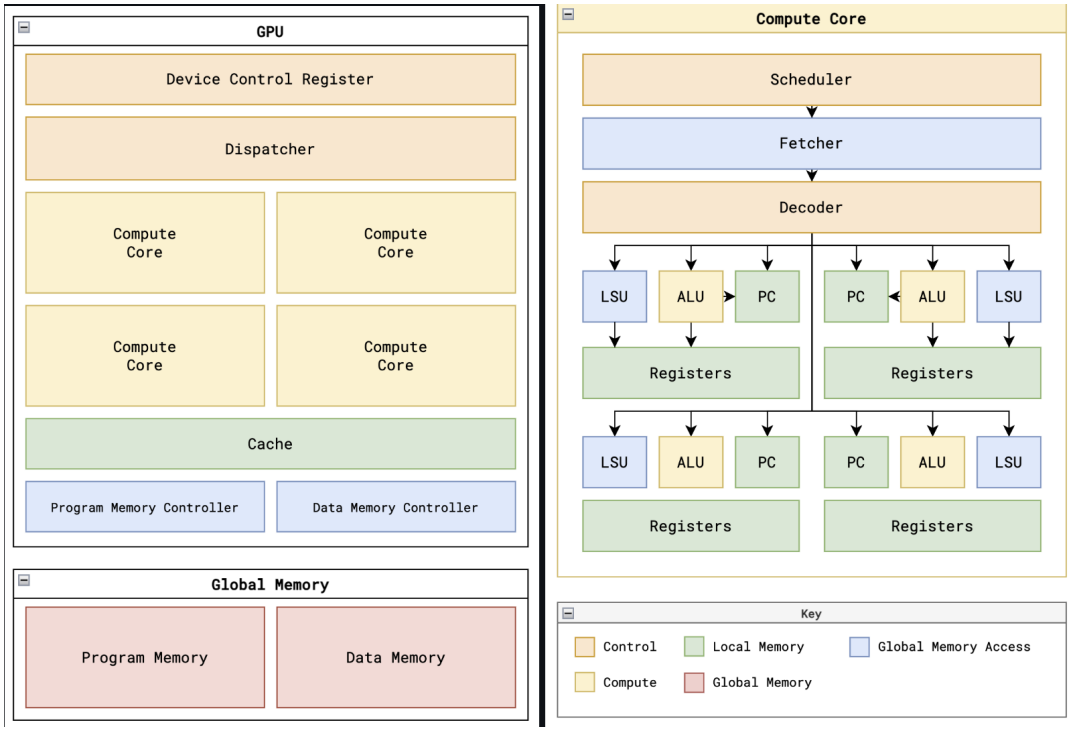


Figure 2: Tiny-GPU system architecture showing data/control flows

3.1.1 Device Control Register

The command center for kernel execution:

- **Registers:**
  - `THREAD_COUNT` (8-bit): Total threads to execute
- **Control Signals:**
  - `start`: Initiates kernel execution
  - `done`: Signals completion to host
  - `reset`: Synchronous reset
- **Operation Sequence:**
  1. Host sets `THREAD_COUNT` register
  2. Host asserts `start` signal
  3. GPU asserts `done` when complete
- **Implementation:**
  - On reset: thread count cleared to zero
  - On clock edge: captures data when write enable is high
  - Output value immediately reflected on thread_count

3.1.2 Dispatcher Unit

The "traffic controller" for thread management:

- **Block Generator:**
  - Creates thread blocks of fixed size (default: 8 threads)
  - Assigns unique Block IDs
  - Calculates total blocks: $\lceil \texttt{THREAD\_COUNT}/\texttt{BLOCK\_SIZE} \rceil$
- **Core Assigner:**
  - Maintains core availability status
  - Sends blocks to available cores
  - Outputs signals: core_start, core_reset, core_block_id
- **Completion Tracker:**
  - Monitors core completion signals (core_done)
  - Signals device controller when all blocks complete
  - Maintains blocks_dispatched and blocks_completed counters
- **FSM States:** RESET, START, RUNNING, DONE

3.2 Memory Subsystem

3.2.1 Program Memory

- **Organization:** 256×16-bit ROM
- **Access Protocol:**
  1. Core provides 8-bit address
  2. Memory outputs 16-bit instruction after 1 cycle
  3. Read-only after initialization

3.2.2 Data Memory

- **Organization:** 256×8-bit RAM
- **Access Protocol:**
  1. Core provides address + write data (for stores)
  2. Controller queues request
  3. Memory responds after fixed latency

3.2.3 Memory Controllers

- **Request Queue:** Stores pending memory requests
- **Arbiter:** Selects next request using round-robin policy
- **Response Handler:** Routes data back to requesting core
- **Channel FSM:** IDLE, READ WAITING, READ RELAYING, WRITE WAITING, WRITE RELAYING
- **Parameters:** ADDR_BITS, DATA_BITS, NUM_CONSUMERS, NUM_CHANNELS

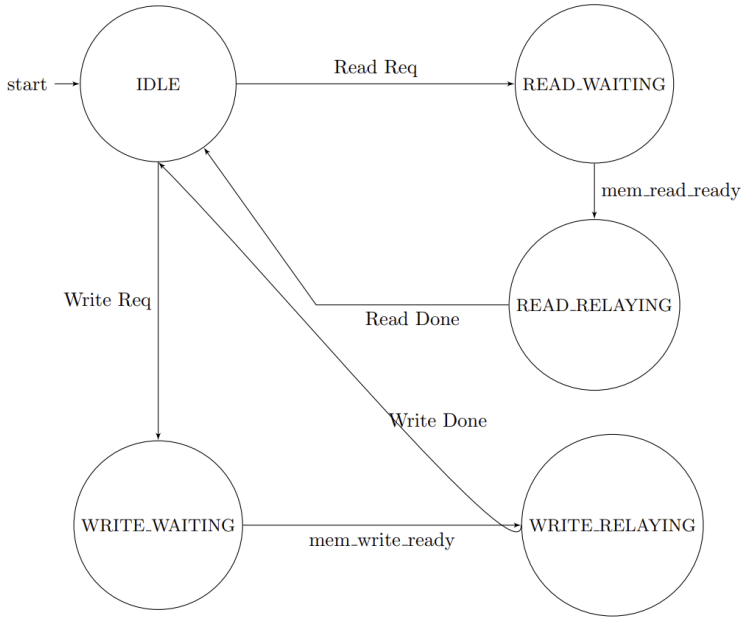


Figure 3: Memory Controller State Transition Diagram

3.3 Compute Core Architecture

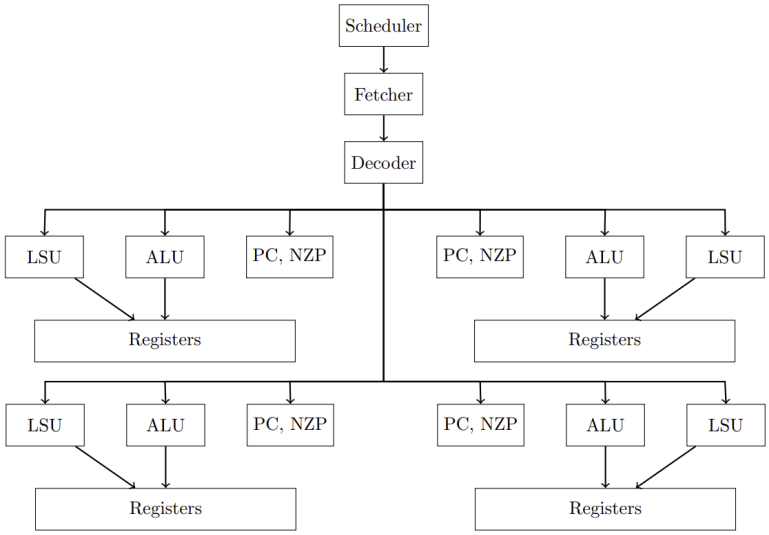


Figure 4: Core Architecture

3.3.1 Scheduler

The core's control unit:

- **States:**
  - IDLE (3'b000), FETCH (3'b001), DECODE (3'b010)
  - REQUEST (3'b011), WAIT (3'b100)
  - EXECUTE (3'b101), UPDATE (3'b110), DONE (3'b111)

- **Operation:**
  - Executes one block to completion before new block
  - Threads execute in lockstep (all threads same instruction)
  - **Stall Logic:** Pauses during WAIT state for memory access
- **Inputs:** clock, reset, start, LSU state, decoded return
- **Outputs:** current PC, scheduler state, done

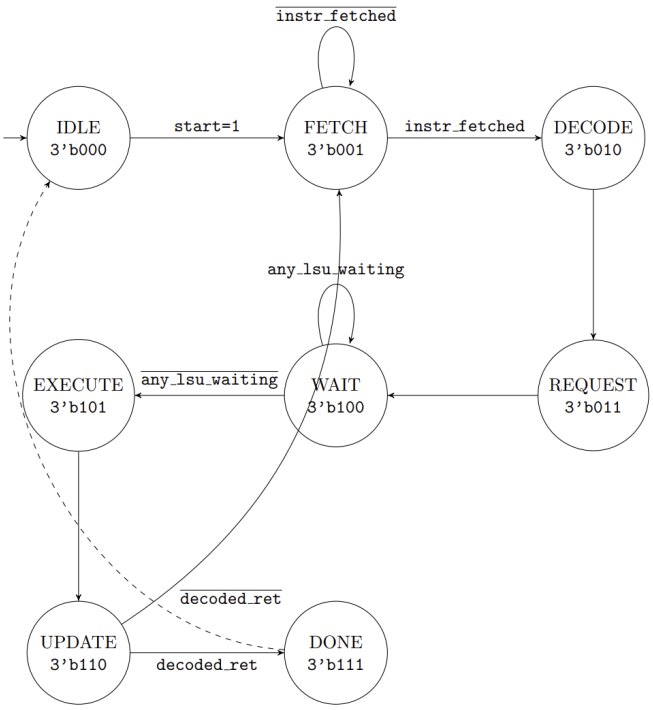


Figure 5: Scheduler State transition diagram

3.3.2 Instruction Fetcher

- **Program Counter (PC):** 8-bit register per thread
- **FSM States:** IDLE, FETCHING, FETCHED
- **Operation:**
  - Asserts mem_read_valid with current PC
  - Waits for mem_read_ready signal
  - Latches instruction on acknowledgment
- **Inputs:** current PC, scheduler state
- **Outputs:** mem_read_address, mem_read_valid, instruction

3.3.3 Instruction Decoder

Converts 16-bit instructions to control signals:

- **Inputs:** clock, reset, core state, instruction
- **Outputs:**
  - Register addresses (RD, RS, RT)
  - Decoded NZP condition
  - Immediate value
  - Control signals: reg_write_enable, mem_read_enable, mem_write_enable
  - ALU control signals

3.3.4 Arithmetic Logic Unit (ALU)

Operation	Function
ADD	Rdest = Rsrc1 + Rsrc2
SUB	Rdest = Rsrc1 - Rsrc2
MUL	Rdest = Rsrc1 × Rsrc2 (lower 8 bits)
DIV	Rdest = Rsrc1 ÷ Rsrc2
CMP	Set flags: Z (zero), N (negative), P (positive)

Table 4: ALU Operations

- **Inputs:** clk, reset, enable, core_state, decoded_alu_arithmetic_mux, decoded_alu_output_mux, rs, rt
- **Outputs:** alu_out (8-bit result or {00000, N, Z, P})
- **Logic:**
  - Outputs NZP flags when decoded_alu_output_mux == 1
  - Performs arithmetic when decoded_alu_output_mux == 0

3.3.5 Load-Store Unit (LSU)

- **Request Handling:**
  1. Send address to memory controller
  2. Wait for acknowledgment
  3. Receive data (for loads)
- **FSM States:** IDLE, REQUESTING, WAITING, DONE
- **Inputs:** clk, reset, enable, core_state, decoded_mem_read_enable, decoded_mem_write_enable, rs, rt
- **Outputs:** lsu_out, lsu_state

3.3.6 Register File

Register	Name	Purpose
R0-R12	General Purpose	Computation storage
R13	%blockIdx	Current block ID
R14	%blockDim	Threads per block
R15	%threadIdx	Thread ID within block

Table 5: Register File Organization

- **Implementation:** 8 threads × 16 registers × 8 bits = 1,024 flip-flops per core
- **Inputs:** block ID, thread ID, threads per block
- **Access:** Two read ports, one write port

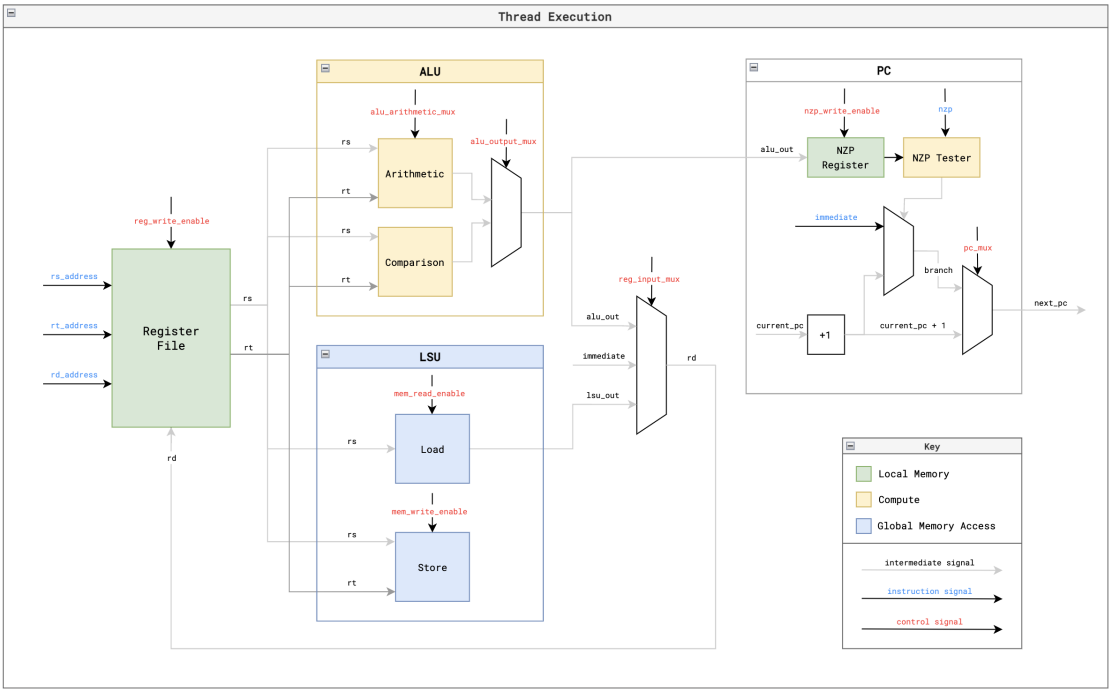


Figure 6: Thread Architecture

4 Instruction Set Architecture

4.1 Instruction Formats

Three instruction formats optimized for common operations

4.2 Instruction Descriptions

Instruction	**Description**
`BR[cond] addr`	Branch to address if condition met (EQ, NE, LT, GT, LE, GE)
`CMP Ra, Rb`	Compare Ra and Rb, set Z/N/P flags
`ADD Rd, Ra, Rb`	Rd = Ra + Rb
`SUB Rd, Ra, Rb`	Rd = Ra - Rb
`MUL Rd, Ra, Rb`	Rd = Ra × Rb
`DIV Rd, Ra, Rb`	Rd = Ra ÷ Rb (integer division)
`LDR Rd, [Ra]`	Load from memory address in Ra to Rd
`STR Rs, [Ra]`	Store Rs to memory address in Ra
`CONST Rd, #imm`	Load 8-bit immediate value into Rd
`RET`	End thread execution
`NOP`	No operation

Table 6: Instruction Set Details

5 Execution Model

5.1 Thread Execution Pipeline

5.1.1 FETCH Stage

1. Scheduler sends PC values to program memory
2. Instruction memory returns 16-bit instructions
3. Fetcher FSM: IDLE → FETCHING → FETCHED

5.1.2 DECODE Stage

- Opcode separated from operands
- Register addresses identified
- Control signals generated

5.1.3 REQUEST Stage

- For memory operations: Send request to controller
- LSU enters REQUESTING state

5.1.4 WAIT Stage

- Pipeline stalled for memory operations
- Wait for memory controller response
- LSU in WAITING state

5.1.5 EXECUTE Stage

- ALU performs arithmetic operations
- Comparison operations set condition flags
- Branch targets calculated

5.1.6 UPDATE Stage

- Results written back to register file
- PC updated for next instruction
- Condition flags stored

5.2 Execution Characteristics

- Single kernel execution at a time
- No branch divergence handling (all threads follow same PC)
- Memory access causes full pipeline stall
- 4 threads per core processed simultaneously

6 Advanced GPU Concepts

6.1 Limitations of Simplified GPUs

Tiny-GPU omits several critical features found in modern GPUs:

- **Multi-layered Caches:** Only implements single cache layer
- **Shared Memory:** No shared memory space for threads
- **Branch Handling:** No support for diverging thread execution
- **Synchronization:** No barrier synchronization mechanism
- **Memory Optimization:** No memory coalescing capability
- **Execution Optimization:** No pipelining or warp scheduling

6.2 Multi-layered Cache & Shared Memory

- **Cache Hierarchy:** Modern GPUs use multiple cache levels
  - Frequently accessed data cached closer to compute units
  - Advanced caching algorithms maximize cache hits
- **Shared Memory:** Dedicated memory space for threads within same block
  - Enables threads to share intermediate results
  - Faster than global memory access

6.3 Memory Coalescing

- **Purpose:** Optimize memory bandwidth utilization
- **Mechanism:**
  - Analyzes queued memory requests
  - Combines neighboring requests into single transaction
  - Example: Multiple threads accessing adjacent matrix elements
- **Benefit:** Reduces addressing overhead and improves efficiency

6.4 Pipelining

- **Purpose:** Maximize resource utilization
- **Mechanism:**
  - Streams execution of multiple instructions simultaneously
  - Maintains sequential execution for dependent instructions
  - Allows continuous execution during memory waits
- **Benefit:** Prevents resource idling during operations

6.5 Warp Scheduling

- **Purpose:** Improve core utilization during stalls
- **Mechanism:**
  - Divides blocks into smaller warps (batches of threads)
  - Executes instructions from one warp while another waits
  - Quickly switches between ready warps
- **Benefit:** Maintains execution throughput during memory access

6.6 Branch Divergence

- **Challenge:** Threads branching to different PCs
- **Modern Solution:**
  - Tracks diverging thread execution paths
  - Manages separate execution streams
  - Handles reconvergence points
- **Tiny-GPU Limitation:** Assumes all threads follow identical PC

6.7 Synchronization & Barriers

- **Purpose:** Coordinate thread execution within blocks
- **Mechanism:**
  - Barrier points where threads wait for peers
  - Ensures all threads reach synchronization point
  - Enables safe data sharing between threads
- **Application:** Essential for shared memory operations

7 Conclusion

This comprehensive analysis has explored GPU architecture through both conceptual foundations and practical implementation of the Tiny-GPU educational processor. The key insights are summarized as follows:

- **Parallel Processing Fundamentals:** Tiny-GPU demonstrates the core SIMD execution model where multiple threads execute identical instructions simultaneously on different data elements, enabling massive parallelism.

- **Architectural Implementation:** The Tiny-GPU design features:
  - Simplified system architecture with device control registers and dispatcher
  - Dedicated memory subsystems with separate program and data memories
  - Compute cores with schedulers, ALUs, and register files
  - Fixed-size thread blocks executing in lockstep
- **Execution Pipeline:** The 6-stage execution model (FETCH-DECODE-REQUEST-WAIT-EXECUTE-UPDATE) processes instructions with:
  - Full pipeline stalls during memory accesses
  - No branch divergence handling
  - Synchronous thread execution
- **Memory Architecture:** Implements:
  - 256×16-bit program memory (ROM)
  - 256×8-bit data memory (RAM)
  - Round-robin memory arbitration
  - Multi-state memory controllers
- **Instruction Set:** The 11-instruction ISA supports core operations including arithmetic, memory access, branching, and thread control.
- **Advanced Concepts:** While Tiny-GPU omits commercial GPU features like:
  - Multi-layered caches and shared memory
  - Memory coalescing and warp scheduling
  - Branch divergence handling and barriers

  It provides a foundation for understanding these enhancements.

Tiny-GPU successfully captures the essential architectural principles of modern GPUs while maintaining simplicity suitable for educational exploration. Its modular design provides a foundation for implementing advanced features like pipelining, cache hierarchies, and warp scheduling to bridge the gap with commercial GPU implementations.