# Coreographer

## Micro-task Scheduled Multi-core RISC-V Architecture

Revised Scope Document

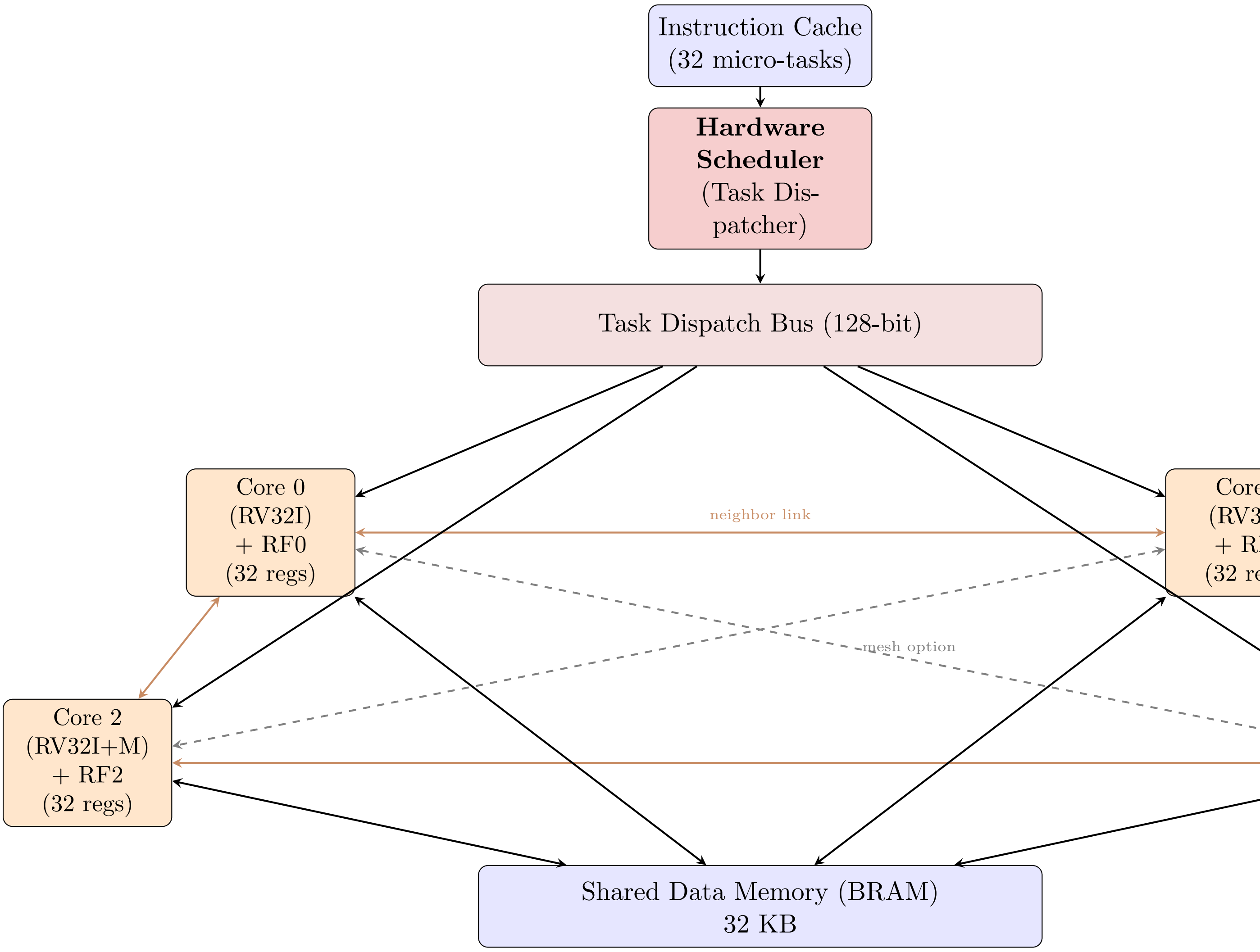
**E. Mihir Divyansh**
EE23BTECH11017

## Executive Summary

This project implements a **heterogeneous multi-core RISC-V architecture** inspired by GPU streaming multiprocessors, featuring clustered cores with local register sharing and a hardware-based micro-task scheduler. The design targets moderately parallel workloads where task-level parallelism can be exploited without the overhead of a full OS scheduler.

## Architecture Overview

1. **Neighbor-based register sharing** replaces global shared register file
    - Each core has its own private 32-register file (RV32I standard)
    - Cores can access registers from neighboring cores via extended instructions
    - Neighbor topology is configurable (And is optionally a performance affecting parameter)
    - **Exploration aspect: (Optional)** Different topologies can be implemented and compared
2. **Parallel hardware scheduler** as dedicated dispatch unit
    - Maintains instruction cache and ready queue for worker cores
    - Broadcasts tasks on shared dispatch bus
    - Cores pull tasks when available
3. **Micro-task programming model**
    - Input programs structured as self-contained micro-tasks. (To be defined)
    - Tasks declare dependencies and resource requirements in headers



*Blue arrows: Neighbor register access — Dashed: Alternative topology*

## System Block Diagram

**Note on Topology:** The diagram shows a ring topology as the baseline. The project will explore different neighbor definitions:

- **Ring:** Each core accesses left/right neighbors (2 neighbors per core)
- **Mesh/Grid:** Each core accesses up/down/left/right neighbors (up to 4 neighbors)
- **Star:** All cores access a central core (1 or 3 neighbors)
- **Custom:** Asymmetric connections based on expected communication patterns

One topology will be chosen for initial implementation, with potential to compare alternatives in stretch goals.

# Detailed Component Specifications

## 1. RISC-V Worker Cores

### ISA Subset:

RV32I base + optional M extension + neighbor register access

- **Cores 0-1:** RV32I only (add, sub, load, store, branches, logical ops)
- **Cores 2-3:** RV32I + M extension (includes mul, mulh, div, rem)
- **All cores:** Extended with neighbor register instructions (see below)
- *Rationale:* Demonstrates heterogeneity and task affinity scheduling

### Pipeline:

3-stage in-order pipeline

- Stage 1: Fetch + Decode
- Stage 2: Execute (ALU / MUL / Address calculation / Neighbor access)
- Stage 3: Memory / Writeback
- Simple stall-on-hazard control (no forwarding initially)
- CPI target: 1.2-1.5 for compute-bound tasks, 2-3 for neighbor access

### Register File Organization:

- Each core has its own private 32-register file (standard 2R1W design)
- **Neighbor Register Access:** Cores can read/write neighbor registers via extended ISA
- **Topology configuration:** Neighbor relationships defined at synthesis time

- **Access latency:** 1-2 additional cycles for neighbor register operations
- No arbitration needed initially (assume neighbors don't conflict)

### Extended ISA for Neighbor Access:

Three new instruction formats for cross-core register communication:

```
NLOAD rd, rs1, neighbor_id # rd = neighbor[neighbor_id].RF[rs1]
NSTORE rs1, rs2, neighbor_id # neighbor[neighbor_id].RF[rs2] = rs1
NMOV rd, rs1, neighbor_id # Atomic:  rd = neighbor.RF[rs1]
```

Where:

- `neighbor_id`: 2-bit field identifying which neighbor (0-3)
- Invalid neighbor_id causes exception or NOP (configurable)
- Operations complete in 2-3 cycles (includes handshake)

### Neighbor Topology Definition:

A configuration register in each core defines its neighbor mapping:

```
NEIGHBOR_MAP[core_id] = {
    neighbor_0: core_id,  // e.g., left neighbor
    neighbor_1: core_id,  // e.g., right neighbor
    neighbor_2: core_id,  // e.g., top neighbor (mesh)
    neighbor_3: core_id   // e.g., bottom neighbor (mesh)
}
```

**Example topologies to explore:**

- **Ring (4 cores):** Core 0 ↔ [Core 1 ↔ Core 3 ↔ Core 2] ↔ Core 0
- **Star (4 cores):** Core 0 (center) ↔ Cores 1, 2, 3 (periphery)
- **Line (4 cores):** Core 0 ↔ Core 1 ↔ Core 2 ↔ Core 3
- **2×2 Mesh:** Each core has up to 4 neighbors (edges have fewer)

## 2. Hardware Scheduler

### Architecture:

Finite State Machine + Control Datapath

### Core Responsibilities:

- **Task queue management:** Maintain FIFO of ready micro-tasks (depth: 32 entries)
- **Dispatch logic:** Select next task based on:
  1. Core availability (idle/busy status from cores)
  2. Task capability requirements (needs multiply? → Cluster 1)
  3. Simple priority or FIFO ordering
- **Dependency tracking** (optional for Phase 2):
  - Maintain 8-bit dependency vector per task
  - Tasks become ready when all dependencies satisfied
  - Cores signal task completion with task ID
- **Instruction cache interface:**
  - Prefetch next 32 tasks from external memory on initialization
  - Simple streaming: no replacement policy needed

### Dispatch Protocol:

1. Scheduler places task on dispatch bus: [Valid — Task_ID — Core_Mask — Instruction_Bundle]
2. Available cores listen on bus
3. Core accepts if: (Core_Mask matches) AND (Core is idle)
4. Core asserts ACK signal
5. Scheduler marks task as dispatched, moves to next

### Performance Counters:

- Total tasks dispatched
- Dispatch stalls (no core available)
- Per-core: tasks executed, cycles busy, cycles idle
- Register file conflicts (stretch goal)

## 3. Micro-Task Format

Each micro-task consists of a **header** (metadata) and a **body** (instructions).

### Task Header Format (32 bits):

```
[31:24] Task ID (8 bits)
[23:16] Dependency vector (8 bits) - IDs of prerequisite tasks
[15:12] Capability flags (4 bits):
        bit 3: Requires multiply (→ must go to Cluster 1)
        bit 2: Requires division
        bit 1: Memory-intensive (hint)
        bit 0: Reserved
[11:8]  Cluster preference (4 bits): 0=any, 1=Cluster0, 2=Cluster1
[7:0]   Instruction count (8 bits): number of instructions in body
```

### Task Body:

- Sequence of 4-16 standard RV32I(M) instructions
- Last instruction must be task completion marker (custom: `TASK_DONE`)
- No branches outside task boundary (tasks are atomic)
- Example:

```
.task 42 # Task ID
.depends 39, 40 # Wait for tasks 39 and 40
.needs_mul # Requires Cluster 1
.instructions
lw r1, 0(r10) # Load A[i]
lw r2, 0(r11) # Load B[i]
mul r3, r1, r2 # Multiply
sw r3, 0(r12) # Store C[i]
TASK_DONE # Signal completion
.end_task
```

# Project Scope and Milestones

## Phase 1: Core System (Weeks 1-6) — MANDATORY

1. **Week 1-2:** Port baseline single-core RISC-V to FPGA
   - Select open-source core (e.g., PicoRV32, SERV, or custom 3-stage)
   - Verify functionality with simple test programs
   - Establish baseline performance metrics
2. **Week 3-4:** Implement neighbor register access mechanism
   - Design neighbor interconnect network (start with simple ring)
   - Implement NLOAD/NSTORE/NMOV instructions in core pipeline
   - Add neighbor configuration registers
   - Test with hand-crafted programs using cross-core register operations

- **Exploration:** Document design decisions and latency trade-offs

3. **Week 5-6:** Design and implement hardware scheduler
   - Implement task queue and dispatch FSM
   - Design dispatch bus protocol
   - Integrate instruction cache (simple ROM/BRAM initially)
   - Unit test: verify task dispatch to idle cores

## Phase 2: System Integration (Weeks 7-9) — MANDATORY

4. **Week 7:** Integrate 4-core system
   - Connect all 4 cores with chosen neighbor topology (ring or mesh)
   - Implement shared memory subsystem (32 KB BRAM)
   - Add performance counters (including neighbor access tracking)
   - Verify neighbor access correctness with synthetic tests
5. **Week 8:** Develop micro-task assembler toolchain
   - Python script to parse task-annotated assembly
   - Generate task headers and instruction bundles
   - Produce memory initialization files for FPGA
6. **Week 9:** First benchmark: Vector dot product
   - Implement as 64 micro-tasks (one per element pair)
   - Verify correctness on FPGA
   - Measure speedup vs single-core baseline

## Phase 3: Evaluation (Weeks 10-12) — TARGET

7. **Week 10:** Second benchmark: Small matrix multiply (8×8 or 16×16)
   - Decompose into row-column dot product tasks
   - Test task affinity (place multiply-heavy tasks on Cores 2-3)
   - **Exploration:** Evaluate neighbor-passing vs memory-passing for intermediate results
8. **Week 11:** Performance analysis
   - Collect metrics: throughput, core utilization, dispatch efficiency
   - Compare against single-core and ideal 4× speedup
   - Identify bottlenecks (register conflicts, scheduler stalls, memory)
9. **Week 12:** Documentation and final demo
   - Prepare FPGA demonstration
   - Write final report
   - Create presentation materials

### Phase 4: Stretch Goals — *OPTIONAL*

- **Topology exploration:** Implement and compare ring vs mesh vs star topologies
  - Measure: neighbor access latency, network congestion, task completion time
  - Identify which workloads benefit from which topology
- Dependency-aware scheduling (allow tasks to declare dependencies)
- Dynamic load balancing (scheduler tracks per-core workload)
- Third benchmark: Stencil computation or image convolution (heavy neighbor communication)
- Visualization tool for task dispatch timeline and neighbor traffic
- Configurable neighbor latency model (explore 1-cycle vs 2-cycle neighbor access)

## Benchmark Selection

**Selection Criteria:**

- Exhibit moderate task-level parallelism (not embarrassingly parallel)
- Fit in on-chip memory (32 KB data limit)
- Simple enough to manually decompose into micro-tasks
- Demonstrate heterogeneous core utilization

### Benchmark 1: Vector Dot Product (Mandatory)

- **Size:** Two 64-element vectors (512 bytes total)
- **Parallelism:** 64 independent multiply-accumulate tasks
- **Why:** Simple, verifiable, demonstrates basic dispatch and cluster affinity
- **Task decomposition:** Each task computes one element: `result[i] = A[i] * B[i]`
- **Expected speedup:** 2.5-3× over single core (limited by reduction phase)

### Benchmark 2: Small Dense Matrix Multiply (Target)

- **Size:** 8×8 or 16×16 matrices (depending on memory constraints)
- **Parallelism:** $N^2$ tasks for N×N output matrix
- **Why:** Compute-bound, stresses register file sharing, benefits from multiply units
- **Task decomposition:** Each task computes one output element via dot product
- **Expected speedup:** 3-3.5× over single core

### Benchmark 3: Streaming Accumulator (Stretch)

- **Size:** 256-element stream, 8-tap accumulator window
- **Parallelism:** Pipeline with overlapping windows
- **Why:** Tests scheduler's ability to maintain steady-state throughput

# Evaluation Metrics

## Performance Metrics

- **Throughput:** Tasks completed per second
- **Speedup:** Performance relative to single-core baseline
- **Core utilization:** % of cycles each core is executing instructions
- **Scheduler efficiency:** Dispatch stalls / total dispatch attempts
- **CPI per core:** Cycles per instruction (measure pipeline efficiency)
- **Neighbor access metrics:**
  - Neighbor register read/write counts per core
  - Average latency per neighbor access
  - Neighbor access efficiency (successful / attempted)
- **Communication patterns:**
  - Neighbor traffic vs memory traffic ratio
  - Hotspot analysis (which neighbor links are most used)

## Resource Utilization (FPGA-specific)

- LUT count and percentage of FPGA capacity
- BRAM usage (register files, memory, instruction cache)
- Maximum clock frequency achieved
- Power consumption estimate (from Vivado/Quartus reports)

## Scalability Analysis (Optional)

- **Topology comparison:** Ring vs mesh vs star performance on different workloads
  - Hypothesis: Mesh excels at stencil workloads, ring better for pipeline patterns
- Vary core count (2, 4, 8 cores) to study neighbor network scaling
- Compare homogeneous (all RV32I) vs heterogeneous (RV32I + RV32IM) configurations
- Analyze neighbor access latency sensitivity (1-cycle vs 2-cycle impact)

## Expected Outcomes

- **Working prototype:** 4-core RISC-V system with hardware scheduler, demonstrated on FPGA
- **Performance gains:** 2.5-3.5× speedup on vector/matrix workloads compared to single-core baseline
- **Architectural insights:** Quantified impact of register file sharing, cluster organization, and task granularity on performance
- **Toolchain:** Micro-task assembler and task decomposition methodology for parallel programs
- **Documentation:** Comprehensive report detailing design decisions, implementation challenges, and experimental results

## Deliverables

1. **RTL codebase:** Synthesizable Verilog/VHDL for all system components
2. **Micro-task toolchain:** Assembler and test program generator (Python scripts)
3. **Benchmark suite:** At least 2 working micro-task programs with verification datasets
4. **FPGA demonstration:** Live demo showing parallel execution and performance counters
5. **Verification artifacts:** Testbenches, simulation waveforms, and functional coverage reports
6. **Performance analysis:** Spreadsheet/graphs comparing baseline vs multi-core performance
7. **Final report:** 15-20 page technical document covering:
    - Architectural design and rationale
    - Implementation details and challenges
    - Experimental methodology
    - Results and analysis
    - Future work and limitations
8. **Presentation:** 15-minute slide deck for final demonstration

# Risk Mitigation

### Risk 1: Multi-ported register file doesn't meet timing

- *Mitigation:* Fall back to time-multiplexed access (2× clock, alternate core access)
- *Alternative:* Reduce to 2R1W and add pipeline stalls

### Risk 2: FPGA resource exhaustion

- *Mitigation:* Reduce to 2 cores (1 cluster) or simplify core pipeline
- *Alternative:* Use smaller register files (16 registers instead of 32)

### Risk 3: Scheduler complexity causes timing violations

- *Mitigation:* Pipeline the scheduler FSM over 2-3 cycles
- *Alternative:* Simplify to pure FIFO dispatch (no capability matching)

### Risk 4: Insufficient time for benchmarking

- *Mitigation:* Focus on Phase 1-2 (working system), use Phase 3 for basic validation only
- *Priority:* Demonstrating functional correctness ¿ extensive performance analysis

# Success Criteria

## Minimum Viable Project (Pass Threshold)

- 4 RISC-V cores successfully integrated on FPGA
- Hardware scheduler correctly dispatches tasks to idle cores
- At least 1 benchmark (vector dot product) runs correctly
- Basic performance counters implemented and verified

## Target Goals (Strong Project)

- All Phase 1-3 milestones completed
- 2 benchmarks demonstrating measurable speedup (2.5×+)
- Heterogeneous cores utilized effectively (task affinity working)
- Comprehensive performance analysis and documentation

## Stretch Goals (Exceptional Project)

- Dependency-aware scheduling implemented
- 3+ benchmarks with detailed performance characterization
- Scalability study (varying cluster count or core count)
- Publication-quality results and visualization

**Project Timeline: 12 Weeks**

Mandatory: Weeks 1-9 — Target: Weeks 10-12 — Stretch: As time allows