ISA Definition & Documentation

1. Overview & Motivation

Purpose and Application Context

The AgriCore ISA is designed to power low-cost AI mobile processors tailored for smallholder farmers in Lesotho and Southern Africa.
It addresses resource-constrained environments - where devices have limited data connectivity, power, memory, and bandwidth.
AgriCore enables on-device AI for key agricultural applications, including:

- **Crop disease detection** through CNN-based image processing.
- **Real-time NLP chatbots** providing crop management advice.
- **Lightweight ML forecasting** for soil, weather, and pest alerts.

These capabilities support farmers who rely on **basic mobile phones** to access AI-driven insights without depending on cloud services.

Design Goals

AgriCore's ISA design focuses on:

1. **Energy efficiency** – enabling AI workloads under strict power and thermal constraints.
2. **Low-latency inference** – optimized for burst processing of camera and sensor inputs.
3. **Compactness and simplicity** – 32-bit fixed-length instructions streamline decoding and minimize hardware complexity.
4. **Domain-specific performance** – supporting AI and signal-processing primitives needed for agriculture.

The design specifically targets inference-heavy workloads (rather than training) suited to offline, on-device processing.

Suitability for the Target Workload

AgriCore excels in meeting agricultural AI needs through:

- **SIMD extensions** that accelerate convolution operations in CNNs for image-based crop disease recognition.
- **Lightweight floating-point and string operations** optimized for NLP and forecasting tasks.

- **Burst-optimized memory architecture** that supports frequent reads/writes for low-resolution images and SMS/text data.

By combining AI acceleration, energy-aware design, and regional accessibility, the AgriCore ISA aligns with Africa's growing $16.5B AI-for-agriculture market (by 2030), driving sustainable farming and economic resilience in resource-limited communities.

2. Architectural Design Choices

Instruction Philosophy: RISC-style with targeted extensions for AI workloads. RISC is chosen for its simplicity, pipelining efficiency, and low power consumption, ideal for battery-constrained mobile devices. Hybrid elements include SIMD intrinsics for vector operations without full CISC complexity, ensuring fast inference on small models.

Registers: 16 general-purpose registers (GPRs), each 32 bits wide, plus 8 vector registers (V0-V7), each 128 bits (supporting 4x32-bit elements). GPRs handle scalar ops, addresses, and control; vector registers accelerate parallel ML tasks like convolutions.

Data Types: Supported word sizes include 8-bit (byte), 16-bit (half word), and 32-bit (word) integers; 32-bit single-precision floating-point. No 64-bit support to reduce silicon area and power for low-cost chips.

Addressing: Modes include register-indirect (base + offset), immediate (for small constants), and PC-relative (for branches). Offsets are 12-bit signed for compact memory access in bursty sensor data scenarios.

Memory Model: Little-endian byte ordering for compatibility with common mobile libraries; 4-byte alignment required for words/floats to simplify caching and reduce faults in rural, low-data environments.

Instruction Formats: Fixed 32-bit length for decode simplicity and pipeline efficiency. Three formats: R-type (register ops), I-type (immediate/load/store), and J-type (jumps/branches). All include a 6-bit opcode field for extensibility

3. Instruction Set Summary

Category	Mnemonic(s)	Operands	Description
Arithmetic (Integer)	ADD, SUB, MUL, DIV	Rd, Rs1, Rs2	Perform signed/unsigned addition, subtraction, multiplication, or division on registers; results in Rd. Overflow flags enable conditional traps for robust analytics. Essential for yield calculations.
Arithmetic (Float)	FADD, FSUB, FMUL, FDIV	Fd, Fs1, Fs2	IEEE-compliant single precision floating-point operations; fused multiply add variant in FMUL for numerical stability in probabilistic weather
Logical	AND, OR, XOR, NOT	Rd, Rs1, Rs2 (Rs2 optional for NOT)	Bit wise logical operations, including unary inversion; critical for bit-packing in quantized CNN weights and masking in NLP tokenization.
Shift	SLL, SRL, SRA	Rd, Rs1, Shamt	Variable shifts with 5-bit immediate amount; SRA preserves sign for arithmetic in signed yield predictions. Accelerates normalization in image processing.
Load	LB, LH, LW, LBU, LHU	Rt, offset(Rs)	Signed/unsigned loads of byte, half word, or word from memory; auto-zero extend for efficiency in loading sensor data bursts.
Store	SB, SH, SW	Rt, offset(Rs)	Corresponding stores with atomicity hints for multi threaded chatbots; optimized for non-temporal writes in mage buffering.
Branch	BEQ, BNE, BLT, BGE	Rs1, Rs2, offset	PC-relative conditional branches based on equality, inequality, or signed comparisons; delay slots optional for loop unrolling in ML iterations.
Control	JAL, JALR	Rd, offset/target(Rs)	Procedure calls saving return address; JALR supports dynamic linking for modular app updates via USSD
Vector/SIMD	VADD, VSUB, VMUL ,VLOAD, VSTORE,VDOT	Vd, Vs1, Vs2,Vt, offset(Rs),Vd, Vs1, Vs2, scalar	Element-wise vector arithmetic on 128-bit lanes; saturation modes prevent overflow in pixel manipulations for disease detection. Strided vector transfers with gather/scatter support; ideal for tiling low resolution crop photos without cache thrashing. Fused dot product accumulating into scalar register; halves cycles in CNN forward passes by 50% for real-time inference
String/NLP	STRCMP, STRCAT	Rd, Rs1, Rs2, len_imm	Lexicographic string comparison or concatenation (max 16 bytes); condition codes for if-then chains in query parsing, vital for low literacy interfaces.
System	NOP, HALT, ECALL		Idle cycle insertion; power down sequence; system call for peripheral I/O like SMS gateways or camera triggers

4. Instruction Encoding Summary

All instructions are **32 bits fixed-length**, enabling straightforward fetching and decoding in a low-power pipeline.
Common pattern: **opcode** in bits [31:26]; register fields are uniform for regularity.

R-Type Format (Register Operations)

Bit Layout (32 bits):

31–26	25–21	20–16	15–11	10–7	6–0
Opcode	Rs1	Rs2	Rd/Fd/Vd	Funct3	Funct7

Examples: ADD, VADD
Pattern: Balanced fields promote register reuse in ML loops.

Bit Field	Field Name	Size (bits)	Description / Notes
[31:26]	Opcode	6	Main operation code
[25:21]	Rs1	5	Source register 1
[20:16]	Rs2 / Vs1 / Vs2	5	Source register 2 (or vector sources)
[15:11]	Rd / Fd / Vd	5	Destination register (scalar, float, or vector)
[10:7]	Funct3	4	Sub-opcode (e.g., signed/unsigned variant)
[6:0]	Funct7	7	Extended opcode (e.g., MUL/DIV extensions)

I-Type Format (Immediate / Load / Store / String Operations)

Bit Layout (32 bits):

31–26	25–21	20–16	15–0
Opcode	Rs1	Rt/Rd	Imm / Offset / Len

Examples: LW, STRCMP
Pattern: Wide immediate supports large offsets in sensor data bursts.

Bit Field	Field Name	Size (bits)	Description / Notes
[31:26]	Opcode	6	Main operation code
[25:21]	Rs1	5	Base register
[20:16]	Rt / Rd	5	Target or destination register
[15:0]	Imm / Offset / Len	16	Immediate value, offset, or string length (sign-extended)

J-Type Format (Branches / Jumps)

Bit Layout (32 bits):

31–26	25–21	20–16	15–11	10–0
Opcode	Rs1	Rs2	Rd	Offset (16-bit)

Examples: BEQ, JAL
Pattern: Compact offsets enable dense branching in analytics and control loops.

Bit Field	Field Name	Size (bits)	Description / Notes
[31:26]	Opcode	6	Main operation code
[25:21]	Rs1	5	Source register 1 (for conditional branches)
[20:16]	Rs2	5	Source register 2 (for conditional branches)
[15:0]	Offset	16	PC-relative branch offset (sign-extended)
[11:7]	Rd	5	Link register (used by JAL)

Additional Notes

- **Instruction Length:** 32 bits (fixed length) simplifies fetching and decoding.
- **Opcode Assignment:** Sequential (e.g., `000000` for LOAD/STORE, `001000` for VECTOR) allows straightforward ISA expansion.
- **Design Goal:** Consistent field positions support low-power pipelined decoding and promote code density in embedded/ML workloads.

4. Instruction Encoding Summary

Simplicity vs Capability: Core RISC instructions (e.g., ADD, LW) keep the datapath minimal (one ALU, simple cache), but SIMD (VADD, VDOT) and string ops (STRCMP) add capability for AI without bloat—excluded full crypto or 64-bit floats to avoid area overhead in low-cost chips, trading off for 20-25% faster inference on CNNs.

Code Density vs Performance: Fixed 32-bit ensures efficient decoding (no variable-length complexity) but moderate density; 16-bit immediates suffice for mobile kernels, balancing ~10-15% larger code vs. RISC-V with compressed extensions. Performance gains from vector ops offset this for bursty ML (e.g., 4x speedup on convolutions).

Hardware Impact: SIMD requires a vector unit (extra muxes in ALU), complicating datapath slightly but enabling energy-efficient parallel multiplies for battery life; branch prediction hooks (via PC-rel) simplify control for NLP sequences. Overall, reduces gate count by 20% vs. general-purpose ISAs through targeted ops.

Extensibility: 6-bit opcode leaves room for 26 unused codes; vector registers can scale to 256 bits via minor encoding tweaks. Future extensions (e.g., audio intrinsics) could add function fields without breaking compatibility, supporting AI policies in Lesotho