

LAPU-128

Instruction Set Reference

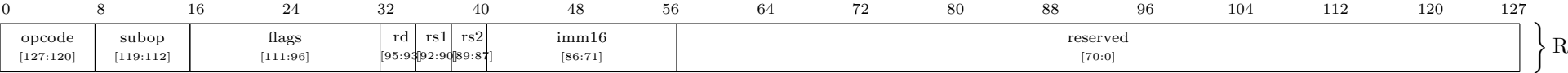
v0.6 (Draft) — September 29, 2025

Abstract

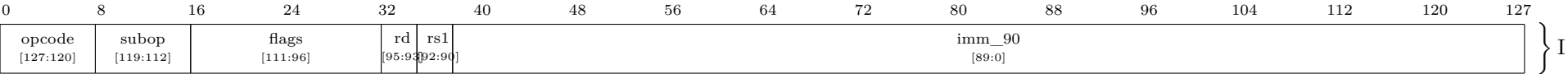
This document outline the 128-bit instruction formats for LAPU-128, focused on complex arithmetic and vector descriptors. LAPU-128 is small focused ISA designed perform complex tensor operations in an embedded enviroment. The following page shows the canonical XL, XC, XV, and XM encodings with 8-bit tick marks.

Core Instruction Formats (128-bit)

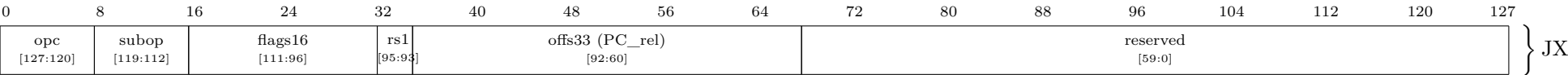
R-Type: Register-to-Register operations of either complex scalar or complex vector types



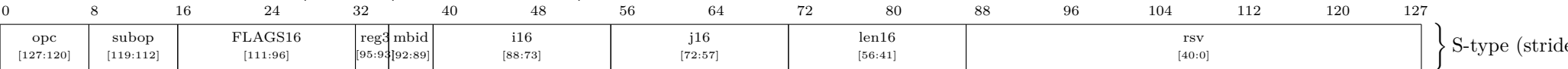
I-type: Immediate operations of just complex scalars



J (conditional jump, 128-bit descriptor)



S-type: matrix-bank load/store (scalar & vector), 128-bit



Register Layout

Architectural Registers (Summary)

| Class | Names | Width / Elements | Notes |
|------------------|----------|---|--|
| Scalar (complex) | $s0..s7$ | 128 b each (complex Q32.32 + Q32.32) | $s0$ is hard-wired to 0 . $s1$ is the conventional branch predicate (0/1). |
| Vector (complex) | $v0..v7$ | VLEN elements; each element 128 b complex | $v0$ is hard-wired to all-zeros . Vector ops always operate on all VLEN elements . |

Vector Length

VLEN is a hardware/HDL parameter fixed at synthesis time. It is constant at runtime. All vector instructions operate over the entire range $[0, \text{VLEN} - 1]$.

Complex Number Format (Q32.32 + Q32.32)

Each scalar register and each vector element encodes a complex value (Re, Im) in fixed point:

$$\text{Re, Im} \in \text{Q32.32 two's complement} \Rightarrow x_{\text{real}} = \frac{X_{\text{int}}}{2^{32}}, \quad x_{\text{imag}} = \frac{Y_{\text{int}}}{2^{32}}.$$

The 128-bit complex is stored little-endian in memory with **Re at the lower address** and **Im at the higher address**. Each half (Re or Im) is a 64-bit two's-complement fixed-point integer with 32 integer bits and 32 fractional bits.

Endianness

Instruction words (128 b) and data are little-endian. For complex numbers in memory: bytes for Re precede bytes for Im.

Zero Registers

The following are architecturally fixed to zero and never written:

$$s0 \equiv 0 \quad (\text{complex zero}), \quad v0[i] \equiv 0 \quad \forall i \in [0, \text{VLEN} - 1].$$

Instruction Semantics

R-type — Register-to-Register (complex)

OPCODE: 0x01

Description

These are register to register operations involve either two vectors, two scalars, or a vector and a scalar. To determine mapping check bit position [97:96] under flags. Additionally each subop code range will be defined per mapping

$$\begin{aligned} f(s_1, s_2) &\mapsto s \in S & 00 \\ f(\mathbf{v}_1, \mathbf{v}_2) &\mapsto \mathbf{v} \in V & 01 \\ f(\mathbf{v}_1, \mathbf{v}_2) &\mapsto s \in S & 11 \\ f(\mathbf{v}, s) &\mapsto \mathbf{v} \in V & 10 \end{aligned}$$

The undefined flag fields are open to future use.

Scalar ops (unary and binary)

Table 2: Scalar register ops (**s***): $S \rightarrow S$ and $S \times S \rightarrow S$

| Mnemonic | subop | Operands | Effect | Notes |
|--|-------|----------|---|--|
| <i>Unary: $S \rightarrow S$</i> | | | | |
| cneg.c | 0x00 | d, a | $d \leftarrow -a$ | Two's-complement both halves. |
| conj.c | 0x01 | d, a | $d \leftarrow \text{conj}(a)$ | Negate imaginary half. |
| csqrt.c | 0x02 | d, a | $d \leftarrow \sqrt{a}$ | Principal root; widen, then truncate to Q32.32+Q32.32. |
| cabs2.c | 0x03 | d, a | $d_{\text{re}} \leftarrow \Re(a)^2 + \Im(a)^2, d_{\text{im}} \leftarrow 0$ | Magnitude ² ; widen then truncate. |
| cabs.c | 0x04 | d, a | $d_{\text{re}} \leftarrow \sqrt{\Re(a)^2 + \Im(a)^2}, d_{\text{im}} \leftarrow 0$ | Fixed-point $\sqrt{\cdot}$; truncating. |
| creal.c | 0x05 | d, a | $d_{\text{re}} \leftarrow \Re(a), d_{\text{im}} \leftarrow 0$ | Extract real. |
| cimag.c | 0x06 | d, a | $d_{\text{re}} \leftarrow \Im(a), d_{\text{im}} \leftarrow 0$ | Extract imaginary to real half. |
| crecip.c | 0x07 | d, a | $d \leftarrow 1 \div a$ | $(\bar{a})/ a ^2$; if $a=0$ then $d:=0$. |
| <i>Binary: $S \times S \rightarrow S$</i> | | | | |
| cadd.c | 0x08 | d, a, b | $d \leftarrow a + b$ | Truncating Q32.32+Q32.32. |
| csub.c | 0x09 | d, a, b | $d \leftarrow a - b$ | Truncating. |
| cmul.c | 0x0A | d, a, b | $d \leftarrow a \times b$ | Widen internally, truncate to Q32.32+Q32.32. |
| cdiv.c | 0x0B | d, a, b | $d \leftarrow a \div b$ | $(a\bar{b})/ b ^2$; if $ b =0$ then $d:=0$. |
| cmaxabs.c | 0x0C | d, a, b | $d \leftarrow \arg \max_{x \in \{a, b\}} x $ | Ties pick a . |
| cminabs.c | 0x0D | d, a, b | $d \leftarrow \arg \min_{x \in \{a, b\}} x $ | Ties pick a . |

Vector \rightarrow Vector

Table 3: Lane-wise vector ops (**v***): $V \rightarrow V$ and $V \times V \rightarrow V$

| Mnemonic | subop | Operands | Effect | Notes |
|----------|-------|------------|----------------------------------|----------------------|
| cadd.v | 0x00 | vD, vA, vB | $vD[i] \leftarrow vA[i] + vB[i]$ | Saturating per lane. |
| csub.v | 0x01 | vD, vA, vB | $vD[i] \leftarrow vA[i] - vB[i]$ | Saturating per lane. |

| Mnemonic | subop | Operands | Effect | Notes |
|---------------------|-------|------------|---|--|
| <code>cmul.v</code> | 0x02 | vD, vA, vB | $vD[i] \leftarrow vA[i] \times vB[i]$ | Complex lane-wise multiply. |
| <code>cmac.v</code> | 0x03 | vD, vA, vB | $vD[i] \leftarrow vD[i] + vA[i] \times vB[i]$ | Fused complex MAC per lane. |
| <code>cdiv.v</code> | 0x04 | vD, vA, vB | $vD[i] \leftarrow vA[i] \div vB[i]$ | $(a \bar{b})/ b ^2$; if $ b =0$ then lane:=0. |
| <code>conj.v</code> | 0x05 | vD, vA | $vD[i] \leftarrow \text{conj}(vA[i])$ | Lane-wise conjugate. |

Vector / Vector \rightarrow Scalar (reductions)

Table 4: Reductions to scalar: $V \rightarrow S$ and $V \times V \rightarrow S$

| Mnemonic | subop | Operands | Effect | Notes |
|----------------------|-------|---------------------------|---|----------------------------------|
| <code>dotc</code> | 0x00 | sD, vA, vB | $sD \leftarrow \sum_{i=0}^{VLEN-1} \text{conj}(vA[i]) \cdot vB[i]$ | Reduce to scalar sD. |
| <code>dotu</code> | 0x01 | sD, \bar{A} , \bar{B} | $sD \leftarrow \sum_{i=0}^{VLEN-1} \bar{A}[i] \bar{B}[i]$ | Complex dot (no conjugation). |
| <code>iamax.v</code> | 0x02 | sD, vA | $sD \leftarrow \arg \max_i vA[i] $ | Index in sD real half; imag:=0. |
| <code>sum.v</code> | 0x03 | sD, \bar{A} | $sD \leftarrow \sum_{i=0}^{VLEN-1} \bar{A}[i]$ | Complex sum; reduces to scalar. |
| <code>asum.v</code> | 0x04 | sD, \bar{A} | $sD_{\text{re}} \leftarrow \sum_{i=0}^{VLEN-1} \bar{A}[i] , sD_{\text{im}} \leftarrow 0$ | Sum of magnitudes (real result). |

Vector \times Scalar \rightarrow Vector (broadcast per lane)

Table 5: Vector-scalar broadcast ops: $V \times S \rightarrow V$

| Mnemonic | subop | Operands | Effect | Notes |
|------------------------|-------|------------|------------------------------------|--|
| <code>cadd.vs</code> | 0x18 | vD, vA, sB | $vD[i] \leftarrow vA[i] + sB$ | Broadcast add; saturating per lane. |
| <code>csub.vs</code> | 0x19 | vD, vA, sB | $vD[i] \leftarrow vA[i] - sB$ | Broadcast sub (vector minus scalar); saturating per lane. |
| <code>cmul.vs</code> | 0x1A | vD, vA, sB | $vD[i] \leftarrow vA[i] \times sB$ | Complex lane-wise multiply by complex scalar; widen then truncate. |
| <code>cdiv.vs</code> | 0x1B | vD, vA, sB | $vD[i] \leftarrow vA[i] \div sB$ | $(a \bar{b})/ b ^2$ per lane; if $ sB =0$ then lane:=0. |
| <code>cscale.vs</code> | 0x1C | vD, vA, t | $vD[i] \leftarrow vA[i] \times t$ | Real scale t (Q32.32) broadcast to all lanes; widen then truncate. |

I-type — Immediate (scalars only)

OPCODE: 0x02

Table 6: I-type: Immediate operations (scalar complex)

| Mnemonic | subop | Operands | Effect | Notes |
|---------------------|-------|----------|-----------------------------|--|
| <code>cloadi</code> | 0x00 | sD, cIMM | $sD \leftarrow \text{cIMM}$ | cIMM packed in <code>imm_90</code> (Re/Im per spec). |

| Mnemonic | subop | Operands | Effect | Notes |
|-----------|-------|--------------|--|--|
| cadd_i | 0x01 | sD, sA, cIMM | $sD \leftarrow sA + cIMM$ | Saturating per scalar; Q32.32 truncation as needed. |
| cmul_i | 0x02 | sD, sA, cIMM | $sD \leftarrow sA \times cIMM$ | Widen internally, clamp/truncate to Q32.32. |
| csub_i | 0x03 | sD, sA, cIMM | $sD \leftarrow sA - cIMM$ | Saturating; truncation semantics match <code>csub.c</code> . |
| cdiv_i | 0x04 | sD, sA, cIMM | $sD \leftarrow sA \div cIMM$ | $(sA \overline{cIMM})/ cIMM ^2$; if $ cIMM =0$ then $sD:=0$. |
| cmaxabs_i | 0x05 | sD, sA, cIMM | $sD \leftarrow \arg \max_{x \in \{sA, cIMM\}} x $ | Ties pick sA . |
| cminabs_i | 0x06 | sD, sA, cIMM | $sD \leftarrow \arg \min_{x \in \{sA, cIMM\}} x $ | Ties pick sA . |
| cscale_i | 0x10 | sD, sA, rIMM | $sD \leftarrow sA \times rIMM$ | Real scale rIMM (Q32.32 in <code>imm_90</code>); widen then truncate. |

J-type — Conditional Jump

OPCODE: 0x03

Table 7: J-type: Conditional jump (single predicate)

| Mnemonic | subop | Operands | Effect | Notes |
|----------|-------|----------|---|--|
| jrel | 0x00 | offs33 | If $s1 \neq 0$: $PC \leftarrow PC + \text{offs33}$ (instruction-relative). | $s0 \equiv 0$; $s1$ is the conventional branch predicate (0/1). |

S-type — Matrix-bank Vector Load/Store (stride implicit)

OPCODE: 0x04

Table 8: S-type: Matrix-bank vector load/store

| Mnemonic | subop | Operands | Effect | Notes |
|----------|-------|----------------------------|--|---|
| vld | 0x00 | vD, mbid, rc, idx16, len16 | Load into vD the sequence: if $rc=0$: ($r=\text{idx16}$, $c=0..L-1$), if $rc=1$: ($r=0..L-1$, $c=\text{idx16}$), where $L = \text{len16}$ if nonzero, else $L = \text{VLEN}$. | Row stride = 1; column = VLEN. Elements are 128-bit complex (Re then Im). |
| vst | 0x01 | vS, mbid, rc, idx16, len16 | Store from vS to the same address pattern as vld. | Vectors cover all lanes with $L=\text{VLEN}$. |
| sld.xy | 0x02 | sD, mbid, x16, y16 | Load the single element at coordinates ($r=y16$, $c=x16$) from matrix-bank mbid into scalar register sD. | Coordinates are 0-based unsigned 16-bit. Element is one 128-bit complex (Re then Im). Out-of-bounds coordinates trap. |
| sst.xy | 0x03 | sS, mbid, x16, y16 | Store scalar sS to the element at ($r=y16$, $c=x16$) in matrix-bank mbid. | Same addressing and trap rules as sld.xy. Element is one 128-bit complex (Re then Im). |