

LAPU-128

Instruction Set Reference

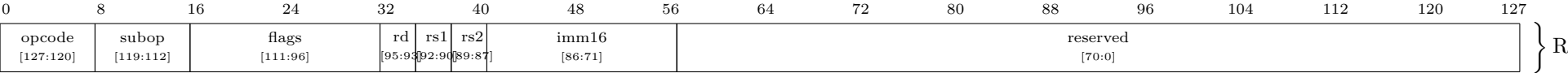
v0.6 (Draft) — November 1, 2025

Abstract

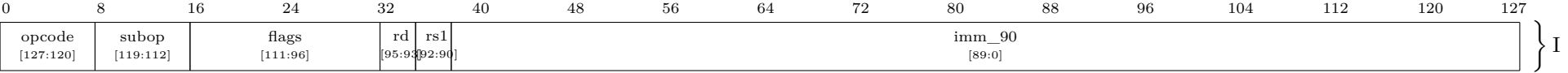
This document outlines the 128-bit instruction formats for LAPU-128, focused on complex arithmetic and vector descriptors. LAPU-128 is a small, focused ISA designed to perform complex tensor operations in an embedded environment. The following page shows the canonical XL, XC, XV, and XM encodings with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

Core Instruction Formats (128-bit)

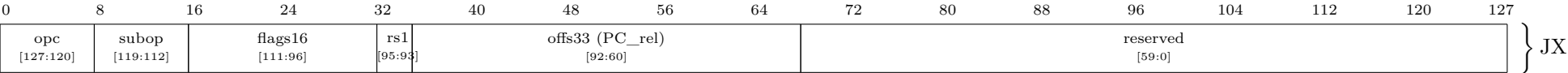
R-Type: Register-to-Register operations of either complex scalar or complex vector types



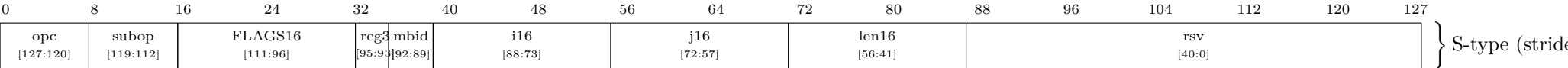
I-type: Immediate operations of just complex scalars



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:

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

Matrix Banks

There will be 4 matrix banks to choose from. Each matrix bank is a square matrix of side N times VLEN where N is some predefined integer greater than 1. The matrix bank will be accessed either through scalar (individual) read/writes or vector read/writes. Vector read/writes will access vectors in either row or major column order and will have no overlap between each other in either mode.

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

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

Encoding notes: The mapping selector lives in `flags[97:96]`. Field `imm16` is currently unused and **must be zero**. All **reserved** bits [70:0] **must be zero**.

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>				
<code>cneg</code>	0x00	d, a	$d \leftarrow -a$	Two's-complement both halves.
<code>conj</code>	0x01	d, a	$d \leftarrow \text{conj}(a)$	Negate imaginary half.
<code>csqrt</code>	0x02	d, a	$d \leftarrow \sqrt{a}$	Principal root; widen, then truncate to Q32.32+Q32.32.
<code>cabs2</code>	0x03	d, a	$d_{\text{re}} \leftarrow \Re(a)^2 + \Im(a)^2, d_{\text{im}} \leftarrow 0$	Magnitude ² ; widen then truncate.
<code>cabs</code>	0x04	d, a	$d_{\text{re}} \leftarrow \sqrt{\Re(a)^2 + \Im(a)^2}, d_{\text{im}} \leftarrow 0$	Fixed-point $\sqrt{\cdot}$; truncating.
<code>creal</code>	0x05	d, a	$d_{\text{re}} \leftarrow \Re(a), d_{\text{im}} \leftarrow 0$	Extract real.
<code>cimag</code>	0x06	d, a	$d_{\text{re}} \leftarrow \Im(a), d_{\text{im}} \leftarrow 0$	Extract imaginary to real half.
<code>crecip</code>	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>				
<code>cadd</code>	0x08	d, a, b	$d \leftarrow a + b$	Truncating Q32.32+Q32.32.
<code>csub</code>	0x09	d, a, b	$d \leftarrow a - b$	Truncating.
<code>cmul</code>	0x0A	d, a, b	$d \leftarrow a \times b$	Widen internally, truncate to Q32.32+Q32.32.
<code>cdiv</code>	0x0B	d, a, b	$d \leftarrow a \div b$	$(a\bar{b})/ b ^2$; if $ b =0$ then $d:=0$.

Vector \rightarrow Vector

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

Mnemonic	subop	Operands	Effect	Notes
<code>vadd</code>	0x00	vD, vA, vB	$vD[i] \leftarrow vA[i] + vB[i]$	Saturating per lane.
<code>vsub</code>	0x01	vD, vA, vB	$vD[i] \leftarrow vA[i] - vB[i]$	Saturating per lane.
<code>vmul</code>	0x02	vD, vA, vB	$vD[i] \leftarrow vA[i] \times vB[i]$	Complex lane-wise multiply.

Mnemonic	subop	Operands	Effect	Notes
vdiv	0x04	vD, vA, vB	$vD[i] \leftarrow vA[i] \div vB[i]$	$(a\bar{b})/ b ^2$; if $ b =0$ then lane:=0.
vconj	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
dotu	0x01	sD, \bar{A} , \bar{B}	$sD \leftarrow \sum_{i=0}^{VLEN-1} \mathcal{A}[i] \mathcal{B}[i]$	Complex dot (no conjugation).
iamax	0x02	sD, vA	$sD \leftarrow \arg \max_i vA[i] $	Index in sD real half; imag:=0.
sum	0x03	sD, \bar{A}	$sD \leftarrow \sum_{i=0}^{VLEN-1} \mathcal{A}[i]$	Complex sum; reduces to scalar.

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

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

Mnemonic	subop	Operands	Effect	Notes
vsadd	0x18	vD, vA, sB	$vD[i] \leftarrow vA[i] + sB$	Broadcast add; saturating per lane.
vssub	0x19	vD, vA, sB	$vD[i] \leftarrow vA[i] - sB$	Broadcast sub (vector minus scalar); saturating per lane.
vsmul	0x1A	vD, vA, sB	$vD[i] \leftarrow vA[i] \times sB$	Complex lane-wise multiply by complex scalar; widen then truncate.
vsdiv	0x1B	vD, vA, sB	$vD[i] \leftarrow vA[i] \div sB$	$(a\bar{b})/ b ^2$ per lane; if $ sB =0$ then lane:=0.

I-type — Immediate (scalars only)

OPCODE: 0x02

Description

`imm_90` is a complex number split into two 45-bit **signed** fixed-point halves (Q22.23, two's complement), packed as:

$$\text{Re} \rightarrow \text{imm_90}[89:45], \quad \text{Im} \rightarrow \text{imm_90}[44:0].$$

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

Mnemonic	subop	Operands	Effect	Notes
<code>cloudi</code>	0x00	sD, cIMM	$sD \leftarrow cIMM$	cIMM packed in <code>imm_90</code> (Re/Im per above).
<code>cadd_i</code>	0x01	sD, sA, cIMM	$sD \leftarrow sA + cIMM$	Saturating per scalar; Q32.32 truncation as needed.
<code>cmul_i</code>	0x02	sD, sA, cIMM	$sD \leftarrow sA \times cIMM$	Widen internally, clamp/truncate to Q32.32.
<code>csub_i</code>	0x03	sD, sA, cIMM	$sD \leftarrow sA - cIMM$	Saturating; truncation semantics match <code>csub.c</code> .
<code>cdiv_i</code>	0x04	sD, sA, cIMM	$sD \leftarrow sA \div cIMM$	$(sA \overline{cIMM})/ cIMM ^2$; if $ cIMM =0$ then $sD:=0$.

J-type — Conditional Jump

OPCODE: 0x03

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

Mnemonic	subop	Operands	Effect	Notes
<code>jrel</code>	0x00	offs33	If $s1 \neq 0$: $PC \leftarrow PC + \text{offs33}$ (instruction-relative).	Encoding: <code>offs33</code> is <i>signed</i> two's complement in <i>instruction units</i> (128-bit words); base is the address of <i>this</i> instruction. Field <code>rs1</code> must be 001b (predicated on <code>s1</code>). All <code>flags16</code> and reserved bits must be zero.

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 $rc=0$: ($r=idx16$, $c=0..L-1$), if $rc=1$: ($r=0..L-1$, $c=idx16$), where $L = len16$ if nonzero, else $L = VLEN$.	Encoding: FLAGS16.rc bit 111 (MSB of FLAGS16). $rc=0$ means row, $rc=1$ means column. Fields idx16 and len16 are unsigned 16-bit . len16=0 selects $L = VLEN$. Row stride = 1; column stride = $VLEN$. 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 .	Same encoding and field conventions as vld .
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 . Mapping: $x16 \rightarrow i16[88:73]$, $y16 \rightarrow j16[72:57]$. Element size 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 trapping rules as sld.xy . Mapping: $x16 \rightarrow i16[88:73]$, $y16 \rightarrow j16[72:57]$.

Global encoding rules. Unless otherwise specified, all **reserved** fields are **must-be-zero** and all currently **unused** fields are encoded as **zero**.