



BRAD DEVICES

Built in Africa, for the World

BradISA V2 Extensions

Compressed ISA, Vector, and Power Management Architecture

Revision 2.1 — July 2026

Confidential — Brad Devices Proprietary

1 V2 Overview

BradISA V2 extends V1 with four major additions:

1. **16-bit compressed instructions** — two 16-bit instructions packed per 32-bit word, reducing code size by $\sim 30\text{--}40\%$
2. **New pipeline models** — Kestrel (8-stage dual-issue in-order) and Falcon-Lite (3-stage ultra-tiny) to span the efficiency range
3. **Vector extension (VSET)** — 32×256 -bit registers, integer/FP elementwise ops, memory ops, reductions, and converts
4. **Power management subsystem** — 24 MSRs, 5 power states, DVFS control, performance counters, clock-gating

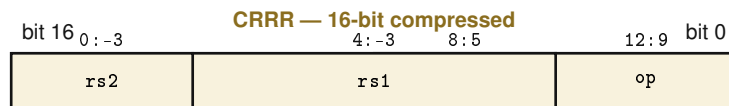
2 Compressed Instruction Format

Compressed mode is activated by setting bit 4 (C) in the STATUS MSR. When C=1, PC advances by 2 bytes (halfword) and each halfword encodes one 16-bit instruction. Two 16-bit instructions pack into a single 32-bit aligned word.

2.1 16-bit Encoding

Table 1: 16-bit compressed instruction format

Format	Bits [15:12]	Bits [11:8]	Bits [7:0]
CRRR	opcode	rd	{rs1[3:0], rs2[3:0]}
CRI	opcode	rd	{rs1[3:0], imm4[3:0]}
CBR	opcode	rs1	offset8[7:0]
CMV	0xC	rd	imm8[7:0]
CNOP	0xD	—	—



Two 16-bit instructions pack per 32-bit word; PC advances 2 bytes

Figure 1: CRRR format — 16-bit compressed register-to-register encoding

2.2 Compressed Opcode Map

Table 2: Compressed opcode mapping

Cop	Mnemonic	Format	Semantics
0x0	C.ADD	CRRR	$rd = rs1 + rs2$
0x1	C.SUB	CRRR	$rd = rs1 - rs2$
0x2	C.AND	CRRR	$rd = rs1 \& rs2$
0x3	C.OR	CRRR	$rd = rs1 rs2$
0x4	C.XOR	CRRR	$rd = rs1 \wedge rs2$
0x5	C.SHL	CRRR	$rd = rs1 \ll rs2[4:0]$
0x6	C.SHR	CRRR	$rd = rs1 \gg rs2[4:0]$
0x7	C.ADDI	CRI	$rd = rs1 + \text{ZEXT}(\text{imm4})$
0x8	C.LDW	CRI	$rd = \text{mem}[rs1 + \text{ZEXT}(\text{imm4}) \times 4]$
0x9	C.STW	CRI	$\text{mem}[rs1 + \text{ZEXT}(\text{imm4}) \times 4] = rd$
0xA	C.BZ	CBR	if $(rs1 == 0)$ PC += SEXT(off8) × 4
0xB	C.BNZ	CBR	if $(rs1 \neq 0)$ PC += SEXT(off8) × 4
0xC	C.MV	CMV	$rd = \text{ZEXT}(\text{imm8})$
0xD	C.NOP	—	No operation
0xE	C.JMP	CBR	PC += SEXT(off8) × 4
0xF	C.CALL	CBR	LR = PC+2; PC += SEXT(off8) × 4

2.3 Code Size Savings

Compressed instructions reduce code size by ~30–40% compared to V1 fixed 32-bit encoding. Common instruction sequences (ADDI, MV, short branches) benefit most. The assembler automatically packs two 16-bit instructions per 32-bit word when **.compress** mode is active.

3 Pipeline Models

3.1 Kestrel — 8-stage Dual-Issue In-Order

Kestrel is a mid-range core bridging the gap between Falcon (single-issue, 5-stage) and Phoenix (octa-issue OoO, 10-stage).

Table 3: Kestrel pipeline characteristics

Property	Value
Pipeline depth	8 stages
Issue width	Dual-issue
Execution	In-order
Branch predictor	Static BTFNT + 16-entry BTB
Load-to-use latency	3 cycles (1 stall)
ALU forwarding	Full (no ALU stall)
Typical IPC	1.2
Peak IPC	1.8
Target DMIPS/MHz	3.0
Target frequency	3.5 GHz
Target TDP	3.5 W
DMIPS/W	3,000

Pipeline Stages

1. **F1** — Instruction address generation, BTB lookup
2. **F2** — Instruction cache read
3. **D1** — Decode instruction 0, register file read
4. **D2** — Decode instruction 1, pairing check
5. **EX1** — ALU 0 / address generation
6. **EX2** — ALU 1 / data memory access
7. **M** — Memory result ready
8. **WB** — Writeback to register file

3.2 Falcon-Lite — 3-stage Ultra-Tiny Core

Falcon-Lite is designed for extreme low-power IoT and embedded applications, targeting <500 LUTs on FPGA.

Table 4: Falcon-Lite pipeline characteristics

Property	Value
Pipeline depth	3 stages
Issue width	Single-issue
Execution	In-order
Register file	8×32 -bit (r0–r7)
Hardware MUL	No (returns 0)
Typical IPC	0.55
Peak IPC	0.70
Target DMIPS/MHz	1.2
Target frequency	1.2 GHz
Target TDP	0.16 W
DMIPS/W	8,800
FPGA LUTs	<500

Pipeline Stages

1. **F** — Fetch instruction, PC increment
2. **D** — Decode, register file read
3. **EX** — Execute, memory access, writeback

3.3 Efficiency Comparison

Table 5: All BradCore pipeline models

Core	Stages	Issue	DMIPS/MHz	DMIPS/W
Falcon-Lite	3	1-wide	1.2	8,800
Falcon	5	1-wide	2.2	10,560
Kestrel	8	2-wide	3.0	3,000
Phoenix C	10	8-wide OoO	5.5	4,714
Phoenix W	10	8-wide OoO	5.5	2,200

4 Vector Extension (VSET)

BradISA V2 adds a 256-bit vector extension (VSET) that closes the SIMD gap with ARM NEON/SVE2, RISC-V Vector 1.0, and Apple AMX for ML/AI, DSP, and media workloads. VSET is the **only** floating-point facility in BradISA V1/V2 — scalar FP registers are not defined; FP scalars live in vector register lane 0 and reductions deliver results through the vector register file. VSET is the fixed-width (AVX-class) standard; the SVE-style variable-length SVEXT path is scoped below as a Gen2 design intent, not a shipped feature.

4.1 Architecture Overview

Table 6: VSET architectural parameters

Parameter	Value
Vector registers	32 (v0–v31), 256-bit each
Vector length (VLEN)	256 bits (fixed)
Element widths (SEW)	8, 16, 32, 64 bits
Elements per register	32×8-bit / 16×16-bit / 8×32-bit / 4×64-bit
FP formats	FP32, FP64 (Gen1); FP16 (Gen2)
Mask register	v0 (reserved for Gen2 predication)
Primary opcode	0xF (VEXT group; 0xF0000000 remains RET)
Vector unit enable	MSR 23 (VSTATUS.VE)
Issue rate	1 vector op/cycle (Phoenix), 256-bit datapath

VSET uses fixed-length 256-bit registers (rather than RISC-V’s software-tunable VLEN) so the vector datapath maps 1:1 onto a 256-bit SIMD lane — the same width as the BradVector GPU cores. A vector register therefore holds 8×FP32, 4×FP64, 8×int32, 16×int16, or 32×int8.



Figure 2: VSET 256-bit register lane layout — SEW selects the active row; all lanes operate in lockstep

4.2 Register File and ABI

Table 7: Vector register ABI conventions

Register	ABI Name	Purpose
v0	vm	Mask register (Gen2 predication); zero-vector convention
v1	va0	Vector argument #0 / return value
v2	va1	Vector argument #1
v3	va2	Vector argument #2
v4	va3	Vector argument #3
v5–v7	vt0–vt2	Caller-saved temporaries
v8–v15	vs0–vs7	Callee-saved
v16–v31	vt3–vt18	Caller-saved temporaries

Callee-saved vector registers **v8–v15** must be preserved across calls, matching the scalar callee-saved set **r8–r12**. The vector calling convention passes the first four vectors in **v1–v4**; FP scalars are passed in lane 0 of **v1–v4**.

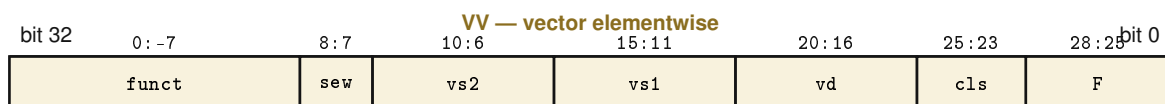
4.3 Encoding

Vector instructions are 32-bit words with primary opcode **0xF**. The fixed encoding **0xF0000000** remains the **V1 RET** instruction for backward compatibility; **all other 0xF** encodings are vector instructions. A 3-bit class field in bits [27:25] selects the functional group.

Table 8: Vector instruction formats

Format	Class	Field layout (32-bit word)
VV	000,001,011,100	F cls[27:25] vd[24:20] vs1[19:15] vs2[14:10] sew[9:8] funct[7:0]
VM	010	F cls[27:25] vd[24:20] base[19:16] mop[15:13] sew[12:11] imm[10:0]
VX	101	F cls[27:25] vd[24:20] vs1[19:15] rs[14:11] sew[10:9] funct[8:0]
VSC	110	F cls[27:25] rd[24:21] vs1[20:16] resv[15:0]

F = primary opcode **0xF** (bits [31:28]). **sew**: 00=8-bit, 01=16-bit, 10=32-bit, 11=64-bit. **vd**, **vs1**, **vs2** select vector registers (5-bit); **base**, **rs**, **rd** select scalar GPRs (4-bit). No vector instruction may encode **0xF0000000**; such a word is always **RET**.

Figure 3: VV format — vector elementwise encoding (F = primary opcode **0xF**)

4.4 Instruction Set

VALU — integer elementwise (class 000, VV)

Table 9: VALU operations

funct	Mnemonic	Operands	Semantics
0x01	V.ADD	vd, vs1, vs2	$vd = vs1 + vs2$ (element, wraps)
0x02	V.SUB	vd, vs1, vs2	$vd = vs1 - vs2$
0x03	V.MUL	vd, vs1, vs2	$vd = vs1 \times vs2$ (low bits)
0x04	V.AND	vd, vs1, vs2	$vd = vs1 \& vs2$
0x05	V.OR	vd, vs1, vs2	$vd = vs1 \mid vs2$
0x06	V.XOR	vd, vs1, vs2	$vd = vs1 \wedge vs2$
0x07	V.SHL	vd, vs1, vs2	$vd = vs1 \ll vs2[\log_2(sew)-1:0]$
0x08	V.SHR	vd, vs1, vs2	$vd = vs1 \gg vs2[\log_2(sew)-1:0]$ (logical)
0x09	V.MIN	vd, vs1, vs2	$vd = \min(vs1, vs2)$ (signed)
0x0A	V.MAX	vd, vs1, vs2	$vd = \max(vs1, vs2)$ (signed)
0x0B	V.CMPEQ	vd, vs1, vs2	$vd = (vs1 == vs2)$ per element (all-ones / 0)
0x0C	V.CMPLT	vd, vs1, vs2	$vd = (vs1 < vs2)$ per element (signed)
0x0D	V.ABS	vd, vs1, —	$vd = vs1 $ per element (signed)
0x0E	V.MOV	vd, vs1, —	$vd = vs1$

VFP — floating-point elementwise (class 001, VV)

Table 10: VFP operations (SEW=32/64; IEEE-754-2019)

funct	Mnemonic	Operands	Semantics
0x01	V.FADD	vd, vs1, vs2	$vd = vs1 + vs2$
0x02	V.FSUB	vd, vs1, vs2	$vd = vs1 - vs2$
0x03	V.FMUL	vd, vs1, vs2	$vd = vs1 \times vs2$
0x04	V.FMADD	vd, vs1, vs2	$vd = vs1 \times vs2 + vd$ (single rounding)
0x05	V.FMIN	vd, vs1, vs2	$vd = \text{fmin}(vs1, vs2)$
0x06	V.FMAX	vd, vs1, vs2	$vd = \text{fmax}(vs1, vs2)$
0x07	V.FCMPEQ	vd, vs1, vs2	$vd = (vs1 == vs2)$ per element
0x08	V.FCMLT	vd, vs1, vs2	$vd = (vs1 < vs2)$ per element
0x09	V.FABS	vd, vs1, —	$vd = vs1 $ per element
0x0A	V.FNEG	vd, vs1, —	$vd = -vs1$ per element
0x0B	V.FSQRT	vd, vs1, —	$vd = \sqrt{vs1}$ per element

FP16 (SEW=16) is a Gen2 option; Gen1 cores treat SEW=8/16 in VFP as an undefined instruction.

VMEM — vector memory (class 010, VM)

Table 11: VMEM operations

mop	Mnemonic	Semantics
000	V.LW	$vd[i] = \text{mem}[\text{base} + \text{sext}(\text{imm11}) + i \times (\text{sew}/8)]$
001	V.SW	$\text{mem}[\text{base} + \text{sext}(\text{imm11}) + i \times (\text{sew}/8)] = vs2[i]$
010	V.LWG	Gather (Gen2) — reserved
011	V.SWG	Scatter (Gen2) — reserved

Number of elements transferred per instruction is $VLEN/\text{sew}$. V.LW uses **vs2** as the destination in the VM layout (**vd** field); V.SW uses **vd** as the store-data vector. The effective base address **base** + **sext**(**imm11**) must be aligned to **sew/8** bytes; a misaligned access raises the unaligned-access exception.

VRED — reductions (class 011, VV)

Reductions read the full source vectors and place the single result in lane 0 of **vd**; all other lanes are zeroed. FP reductions accumulate in the selected FP precision with single rounding at the end.

Table 12: VRED operations

funct	Mnemonic	Operands	Semantics
0x01	V.SUM	vd , vs1 , —	$vd[0] = \Sigma vs1$ (integer, SEW)
0x02	V.FSUM	vd , vs1 , —	$vd[0] = \Sigma vs1$ (FP32/FP64)
0x03	V.MAX	vd , vs1 , —	$vd[0] = \max(vs1)$ (signed)
0x04	V.MIN	vd , vs1 , —	$vd[0] = \min(vs1)$ (signed)
0x05	V.FMAX	vd , vs1 , —	$vd[0] = \text{fmax}(vs1)$
0x06	V.FMIN	vd , vs1 , —	$vd[0] = \text{fmin}(vs1)$
0x07	V.DOT	vd , vs1 , vs2	$vd[0] = \Sigma vs1[i] \times vs2[i]$ (int, accumulate $2 \times \text{SEW}$)
0x08	V.FDOT	vd , vs1 , vs2	$vd[0] = \Sigma vs1[i] \times vs2[i]$ (FP, single rounding)

V.DOT is the int8 ML workhorse: at SEW=8 it computes a 32-lane dot product into a 32-bit result in one instruction.

VCVT — convert (class 100, VV)

Table 13: VCVT operations

funct	Mnemonic	Operands	Semantics
0x01	V.CVT.SF	vd , vs1 , —	signed int $\rightarrow$ FP (round-to-nearest)
0x02	V.CVT.FS	vd , vs1 , —	FP $\rightarrow$ signed int (round toward zero)
0x03	V.CVT.UF	vd , vs1 , —	unsigned int $\rightarrow$ FP
0x04	V.CVT.FU	vd , vs1 , —	FP $\rightarrow$ unsigned int
0x05	V.WIDEN	vd , vs1 , —	$vd:vd+1 = \text{widen}(vs1)$; vd even
0x06	V.NARROW	vd , vs1 , —	$vd[127:0] = \text{narrow}(vs1)$

V.WIDEN doubles the element width ($\text{SEW} \rightarrow 2 \times \text{SEW}$) across a register pair: **vd** holds elements $[0, N/2-1]$, **vd+1** holds $[N/2, N-1]$. **vd** must be even. V.NARROW halves the element width; the

result occupies the low 128 bits of **vd**.

VSPL — scalar-vector (class 101, VX)

Table 14: VSPL operations (**rs** = scalar GPR)

funct	Mnemonic	Operands	Semantics
0x01	V.BCAST	vd, —, rs	vd = broadcast(rs) to all lanes
0x02	V.ADDX	vd, vs1, rs	vd = vs1 + rs
0x03	V.SUBX	vd, vs1, rs	vd = vs1 - rs
0x04	V.MULX	vd, vs1, rs	vd = vs1 × rs
0x05	V.FADDX	vd, vs1, rs	vd = vs1 + rs (FP)
0x06	V.FMULX	vd, vs1, rs	vd = vs1 × rs (FP)
0x07	V.INS	vd, vs1, rs	vd = vs1 with lane 0 = rs

For $SEW < 32$, **rs** is truncated to **sew** bits before broadcast; for $SEW=64$ it is sign-extended.

VSC — scalar extract (class 110, VSC)

Table 15: VSC operation

funct	Mnemonic	Operands	Semantics
0x01	V.EXT	rd, vs1, —	rd = vs1 lane 0 (zero/sign-extended to 32 bits)

V.EXT is the scalar read path that pairs with reductions and V.INS.

4.5 Vector Status MSR

Table 16: MSR 23 — VSTATUS

Bit	Flag	Description
0	VE	Vector unit enable. 0 $\Rightarrow$ any vector instruction traps (undefined instruction, cause 1)
1–4	resv	Reserved
5	VILL	Sticky: illegal vector encoding executed
6	VFP	Sticky: FP exception flag (NaN produced, overflow)

VILL and VFP are sticky flags written by the hardware and cleared only by software (write 1 to clear). Gen1 FP never traps; NaN is propagated per IEEE-754-2019 and VFP is set.

4.6 Example — SAXPY

The classic single-precision $y[i] = a \cdot x[i] + y[i]$ loop, 8 elements per vector operation:

```

1      ; r1=x, r2=y, r3=n, r4=&a (8xFP32 per register)
2  saxpy:
3      V.BCAST v2, --, r4      ; v2 = a broadcast
4      V.LW    v3, r1, 0       ; v3 = x[0..7]
5      V.FMUL  v3, v3, v2      ; v3 = a * x
6      V.LW    v4, r2, 0       ; v4 = y[0..7]
7      V.FADD  v3, v3, v4      ; v3 = a*x + y

```

```

8      V.SW      r2,  v3, 0          ; y[0..7] = v3
9      ADDI      r1, r1, 32          ; advance 8 floats
10     ADDI      r2, r2, 32
11     ADDI      r3, r3, -8
12     BNZ       r3, saxpy
13     RET

```

4.7 Autovectorization Notes

- Clang/LLVM target hooks map <8 x float>, <4 x double>, <8 x i32>, <16 x i16>, <32 x i8> vectors directly onto one register — no lane splitting or permutation code needed.
- Tail handling uses the scalar core: remaining elements use the **r**-register ALU after the **V.*** loop finishes.
- **V.FMADD** and **V.FDOT** preserve the LLVM **contract** fast-math contract, enabling fused multiply-add codegen.
- int8 ML kernels map **V.DOT** from LLVM reduction intrinsics and `@llvm.fmuladd` on <32 x i8>.
- **VSTATUS.VE** is set once by the OS at context switch; the vector register file is part of the architectural state and saved/restored by the kernel, preserving **v8–v15** per ABI.

4.8 Scopeable Vector Extension (SVEXT) — [Gen2 Planning]

On the question of AVX vs. SVE equivalents: **VSET** is the **AVX counterpart** — a fixed 256-bit SIMD standard with one datapath width and stable code generation. That is deliberate and stays.

SVE-style *variable-length* vector support (SVEXT) is **planned for the Gen2 window**, not shipped now, for three honest reasons: scalar/VSET code covers today's shipping software; a variable-length VLA model changes the ABI and toolchain in ways the Gen1 compiler/OS stack is not ready for; and the chip area is better spent on the BradVector GPU engines that already absorb wide-SIMT workloads. SVEXT is therefore a *design intent*, explicitly not yet a committed feature, so no spec is fabricated here. Intended Gen2 shape (subject to change):

- **VLEN** 128–512 bit, variable length per processor (SVE-style vector-length agnostic code, **PTRUE/WHILELT**-class vector-length predication).
- Same **v0** register file uplifted to dynamic lengths, keeping the **VSET** ABI as the portable baseline and letting **SVL** be probed via a new **VSTATUS.SVL** field or an **ID_VECTOR** MSR.
- Narrow (video/DSP) code stays on fixed **VSET**; HPC/AI loops compile to **SVEXT** where the **VLEN** pays for itself.

5 Power Management Subsystem

5.1 New MSRs

BradISA V2 adds 17 new MSRs — 16 for power management and performance monitoring plus MSR 23 (VSTATUS) for the vector extension:

Table 17: V2 MSR additions

Index	Mnemonic	Description
8	PM_CTRL	Power management control
9	PSTATE	Current P-state
10	PSTATES	Available P-states bitmap
11	VOLTAGE	Core voltage in mV
12	FREQ	Core frequency in MHz
13	ENERGY	Accumulated energy in μJ
14	PWR_CAP	Power cap in mW
15	TEMP	Die temperature in $^{\circ}\text{C}$
16	PERF_CNT0	Performance counter 0
17	PERF_CNT1	Performance counter 1
18	PERF_CNT_CTRL	Perf counter control
19	L2_PART	L2 cache partitioning
20	L3_PART	L3 cache partitioning
21	CG_MASK	Clock-gating unit mask
22	DBG_CTRL	Debug control
23	VSTATUS	Vector unit status (VE, VILL, VFP)

5.2 Power States

The PM_CTRL register encodes the current C-state in bits [3:0]:

Table 18: Power C-states

State	Code	Name	Behaviour
C0	0	Active	Full operation
C1	1	WFI	Clock-gated, instant wake
C1E	2	WFI + Vmin	Voltage reduced by 20%
C2	3	Sleep	Power-gated, cache retained
C2E	4	Sleep+	Power-gated, cache flushed
C3	5	Deep sleep	Full power-down, wake from IRQ

5.3 Status Register V2 Flags

Table 19: New STATUS register bits

Bit	Flag	Description
4	C	Compressed mode enable
5	EE	Energy-efficient mode hint
6	DBG	Debug mode

5.4 Performance Counters

PERF_CNT_CTRL configures two 32-bit counters (PERF_CNT0, PERF_CNT1), assigning one event type to each:

Table 20: Performance event selectors

Code	Event
0	CPU cycles
1	Instructions retired
2	Branches
3	Branch mispredicts
4	Load-use stalls
5	L1 cache misses
6	L2 cache misses
7	L3 cache misses
8	Stall cycles
9	IPC

5.5 Clock-Gating Mask

The CG_MASK register (MSR 21) controls per-unit clock gating. Each bit corresponds to a pipeline unit:

Table 21: Clock-gating unit mask

Bit	Unit
0	Fetch
1	Decode
2	Rename
3	Dispatch
4	Issue
5	Execute
6	Memory
7	Writeback

6 Branch Predictor (Phoenix Optimisation)

Phoenix can be configured with a tournament branch predictor:

Table 22: Branch predictor configuration

Parameter	Value
Default type	Tournament (gshare + bimodal)
BTB size	256 entries
BTB associativity	4-way
Global history bits	12
Alternative types	None, Static BTFNT, 2-bit saturating

6.1 Load-Forwarding

Phoenix supports load-to-use forwarding when the load result is consumed within 2 cycles of the load instruction, reducing the load-use penalty from 3 cycles to 1.

7 Revision History

Revision	Date	Changes
1.0	July 2026	Initial BradISA V1 specification — Kinetic Gen1
2.0	July 2026	Added V2 compressed instructions, Kestrel pipeline, Falcon-Lite pipeline, power management MSRs, branch predictor spec, performance counters
2.1	July 2026	Added Vector Extension (VSET): 32×256 -bit registers, integer/FP elementwise ops, memory ops, reductions, converts, VSTATUS MSR 23
2.2	July 2026	Clarified VSET as the fixed-width (AVX-class) standard; scoped SVEXT (SVE-style variable-length) as a Gen2 design intent, not a shipped feature