1. Harmonising the RV Vector Extension with Andes Packed SIMD proposal

Provided the Andes "Packed SIMD" proposal is restricted to 14 vectors of 8 bit elements and 14 vectors of 16 bit elements, it can be made into a forward compatible subset of the RV Vector (RVV) extension. (For an overview of the RVV instruction set, see slide on the next page). The "Harmonised" RVV/Packed SIMD (RVP) proposal would work as follows:

1. MVL, setvl instruction & VL CSR work as per RVV. VLD and VST are supported, but RVP implementations may choose to load/store to/from Integer register file (rather than from a dedicated Vector register file). VLD and VST in this case will have similar behaviour to LW/LD and SW/SD respectively, but only operate on up to VL elements (see point #4 below). Mapping of v0-31 <-> r0-31 IS FIXED AT 1:1. (An exception may be made to map v1 to r5, as otherwise may clash with procedure linkage).

Note (i): Thus, RVP implementations have a choice of providing a dedicated Vector register file, or sharing the integer register file, but not both simultaneously. (Supporting both would need a CSR mode switch bit).
Note (ii): If integer register file is used for vector operations, any callee saved registers (r2-4, 8-9, 18-27) must be saved with RVI SW or SD instructions, before being used as vector registers (this register saving behaviour is harmless but redundant when RVP code is run on a machine with a dedicated vector reg file).

2. VLDX, VSTX, VLDS, VSTS are not supported in RVP (ie: traps, to keep RVP implementations simple)

3. In the absence of an explicit VDCFG setup, the vector registers (when shared with Integer register file) are to default into two "banks" as follows:
   v0-v15: vectors with INT8 elements, split into 8 x signed (v0-v7) & 8 x unsigned (v8-v15)
   v16-v29: vectors with INT16 elements, split into 8 x signed (v16-v23) & 6 x unsigned (v24-v29)

Having the above default vector type configuration harmonises most of the Andes SIMD instruction set (which explicitly encodes INT8 vs INT16 vector types as separate instructions). The main change from the Andes SIMD proposal is that instructions are restricted to 14 registers of each vector element type (with element size explicitly encoded in the most significant bit of the 5 bit register specifier fields).

Note (i): To preserve forward RVV compatibility, programmers should still explicitly setup VDCFG to the above default vector types
Note (ii): Essentially the same register allocation algorithm used for RVV can be used for RVP, except the algorithm should preferentially use temporary registers first, before using saved registers
Note (iii): v30-v31 are reserved for 32 bit operations (see Section 2.3 of this document), and hence not part of the register bank of INT16 vectors.

4. The default RVV MVL value (in absence of explicit VDCFG setup) is to be MVL = 2 on RV32I machines and MVL = 4 on RV64I machines.

However, note RV32I registers can fit 4x INT8 elements. To preserve Andes SIMD behaviour, all VOP instructions should still operate on all "unused" elements in the register, regardless of MVL. (This is still compliant with the RVV spec, provided elements from VL..MVL-1 are set to zero). VMEM instructions however will only operate on VL elements, and so where full Andes SIMD compliance is required (without RVV forward compatibility), LW/LD and SW/SD are to be used instead of VLD and VST.

5. A programmer can configure VDCFG with the any mix of these alternative configurations:
   (i): v0-v31 are all INT 16, and MVL is same as point #4 above
   (ii): v0-v31 are all INT 8 and MVL is 4 on RV32I and 8 on RV64I
   (iii): A lesser number of registers (<v31) could be supported, eg. default is only v0-v29 defined. (Accessing registers beyond maximum defined by VDCFG is to be legal, with a type of INT32 assumed. However, this is not to affect the MVL, which is to be calculated based on INT8/INT16 vectors only)
   (iv): With the above alternative configs, there can be any split between signed & unsigned.

The above are pure subsets of valid RVV VDCFG configurations (and hence forward compatible between RVP and RVV, whilst also keeping RVP simple). Other useful element types are fixed point fraction types and small integer(4 bit to 7 bit) elements. However these are omitted for now as they aren't currently part of RVV spec, and the intention of this proposal is to harmonise the Andes SIMD instructions into a subset of RVV.

Overview of instruction set and encoding of the RV Vector Extension (RVV): (Taken from Roger Espaza's Dec 2017 presentation)

31 30 29 28 27	26	25	24 23 22 21 20	19 18 17 16 15	14	13	12	11 10 9 8 7	6 5 4 3 2 1 0	
src3	n	sub	src2	src1	3s	m	m	dest	OPCODE	Example
vs3	0	0	vs2	vs1	1	m	m	vd	VOP	vmadd
func6		i	src2	src1	3s	m	m	dest	OPCODE	Example
func6		0	vs2	vs1	0	m	m	vd	VOP	vadd
func6		0	0	vs1	0	m	m	vd	VOP	vsqrt
func6		0	new dest type	vs1	0	m	m	vd	VOP	vcvt
func6		0	rs2	rs1	0	m	m	vd	VOP	vmov.v.x vd[rs2] = rs1
func6		0	rs2	vs1	0	m	m	xd	VOP	vmov.x.v xd = vs1[rs2]
func3 imm		1	imm	vs1	0	m	m	vd	VOP	vaddi
imm	op		src2	src1	op	m	m	dest	OPCODE	Example
imm	0	0	imm	rs1	0	m	m	vd	VMEM	vld
imm	0	0	imm	rs1	1	m	m	vs1	VMEM	vst
imm	0	1	rs2	rs1	0	m	m	vd	VMEM	vlds
imm	0	1	rs2	rs1	1	m	m	vs1	VMEM	vsts
imm	1	0	vs2	rs1	0	m	m	vd	VMEM	vldx
imm	1	0	vs2	rs1	1	m	m	vs1	VMEM	vstx
func3 a r	1	1	vs2	rs1	1	m	m	vd	VMEM	vamoadd

Instruction Summary for "P" ISA Extension Proposal

Document Number xxxxxxxxx

Date Issued 2017-11-23

2. DSP ISA Extension Instruction Summary

2.1. Shorthand Definitions

- r.H == rH1 → r[31:16], r.L == r.H0 → r[15:0]
- r.B3 → r[31:24], r.B2 → r[23:16], r.B1 → r[15:8], r.B0 → r[7:0]
- r[xU] → the upper 32-bit of a 64-bit number; xU represents the GPR number that contains this upper part 32-bit value.
- r[xL] → the lower 32-bit of a 64-bit number; xL represents the GPR number that contains this lower part 32-bit value.
- r[xU].r[xL] → a 64-bit number that is formed from a pair of GPRs.
- s>> → signed arithmetic right shift
- u>> → unsigned logical right shift
- SAT.Qn() → Saturate to the range of $[-2^n, 2^n-1]$, if saturation happens, set PSW.OV.
- SAT.Um() → Saturate to the range of $[0, 2^m-1]$, if saturation happens, set PSW.OV.
- RUND() → Indicate "rounding", i.e., add 1 to the most significant discarded bit for right shift or MSW-type multiplication instructions.
- Sign or Zero Extending functions:
  - SEm(data) → Sign-Extend data to m-bit.
  - ZEm(data) → Zero-Extend data to m-bit.
- ABS(x) → Calculate the absolute value of "x".
- CONCAT(x,y) → Concatinate "x" and "y" to form a value.
- u< → Unsinged less than comparison.
- u<= → Unsinged less than & equal comparison.
- u> → Unsinged greater than comparison.
- s* → Signed multiplication.
- u* → Unsigned multiplication.
- rt is Rd in RISC-V ISA terminology.
- ra is Rs1 and rb is Rs2 in RISC-V ISA terminology.

2.2. SIMD Data Processing Instructions

2.2.1. 16-bit Addition & Subtraction Instructions

Proposed encoding of these instructions as follows (assuming "default" VDCFG CSR setting):
ADD16 -> VADD rt[16-29], ra[16-29], rb[16-29] with VOP's mm bits = 0
SUB16 -> VSUB rt[16-29], ra[16-29], rb[16-29] with VOP's mm bits = 0

NOTES:
1. K (Halving) and S (Saturation) signed/unsigned variants to be encoded by a combination of the following:
(a) dividing register bank into signed (r16-23) and unsigned (r24-29) sub-banks
(b) using of VOP"s "mm" field, to override global Saturation CSR mode bit
(mm=01 is unused by V extension, and could be used to force Saturation for the instruction
(c) using VOP's FP/unused opcodes for Halving variants, which only make sense for integer types (eg. VROUND presumably is not needed for integer types?)

2. Alternative VDCFG CSR settings can provide other desired register splits between signed/unsigned and 8/16 bit int

Mnemonic	Instruction	Operation
ADD16 rt, ra, rb	16-bit Addition	rt.H*x* = ra.H*x* + rb.H*x*; (*x*=1..0)
RADD16 rt, ra, rb	16-bit Signed Halving Addition	rt.H*x* = (ra.H*x* + rb.H*x*) s>> 1; (*x*=1..0)
URADD16 rt, ra, rb	16-bit Unsigned Halving Addition	rt.H*x* = (ra.H*x* + rb.H*x*) u>> 1; (*x*=1..0)
KADD16 rt, ra, rb	16-bit Signed Saturating Addition	rt.H*x* = SAT.Q15(ra.H*x* + rb.H*x*); (*x*=1..0)
UKADD16 rt, ra, rb	16-bit Unsigned Saturating Addition	rt.H*x* = SAT.U16(ra.H*x* + rb.H*x*); (*x*=1..0)
SUB16 rt, ra, rb	16-bit Subtraction	rt.H*x* = ra.H*x* - rb.H*x*; (*x*=1..0)
RSUB16 rt, ra, rb	16-bit Signed Halving Subtraction	rt.H*x* = (ra.H*x* - rb.H*x*) s>> 1; (*x*=1..0)

Mnemonic	Instruction	Operation
URSUB16 rt, ra, rb	16-bit Unsigned Halving Subtraction	rt.H*x* = (ra.H*x* - rb.H*x*) u>> 1; (*x*=1..0)
KSUB16 rt, ra, rb	16-bit Signed Saturating Subtraction	rt.H*x* = SAT.Q15(ra.H*x* - rb.H*x*); (*x*=1..0)
UKSUB16 rt, ra, rb	16-bit Unsigned Saturating Subtraction	rt.H*x* = SAT.U16(ra.H*x* - rb.H*x*); (*x*=1..0)
CRAS16 rt, ra, rb	16-bit Cross Add & Sub	rt.H = ra.H + rb.L; rt.L = ra.L – rb.H;
RCRAS16 rt, ra, rb	16-bit Signed Halving Cross Add & Sub	rt.H = (ra.H + rb.L) s>> 1; rt.L = (ra.L – rb.H) s>> 1;
URCRAS16 rt, ra, rb	16-bit Unsigned Halving Cross Add & Sub	rt.H = (ra.H + rb.L) u>> 1; rt.L = (ra.L – rb.H) u>> 1;
KCRAS16 rt, ra, rb	16-bit Signed Saturating Cross Add & Sub	rt.H = SAT.Q15(ra.H + rb.L); rt.L = SAT.Q15(ra.L – rb.H);
UKCRAS16 rt, ra, rb	16-bit Unsigned Saturating Cross Add & Sub	rt.H = SAT.U16(ra.H + rb.L); rt.L = SAT.U16(ra.L – rb.H);
CRSA16 rt, ra, rb	16-bit Cross Sub & Add	rt.H = ra.H - rb.L; rt.L = ra.L + rb.H;
RCRSA16 rt, ra, rb	16-bit Signed Halving Cross Sub & Add	rt.H = (ra.H - rb.L) s>> 1; rt.L = (ra.L + rb.H) s>> 1;
URCRSA16 rt, ra, rb	16-bit Unsigned Halving Cross Sub & Add	rt.H = (ra.H - rb.L) u>> 1; rt.L = (ra.L + rb.H) u>> 1;
KCRSA16 rt, ra, rb	16-bit Signed Saturating Cross Sub & Add	rt.H = SAT.Q15(ra.H - rb.L); rt.L = SAT.Q15(ra.L + rb.H);
UKCRSA16 rt, ra, rb	16-bit Unsigned Saturating Cross Sub & Add	rt.H = SAT.U16(ra.H - rb.L); rt.L = SAT.U16(ra.L + rb.H);

2.2.2. 8-bit Addition & Subtraction Instructions

Proposed encoding of these instructions as follows (assuming "default" VDCFG CSR setting):
ADD8 -> VADD rt[0-15], ra[0-15], rb[0-15] with VOP's mm bits = 0
SUB8 -> VSUB rt[0-15], ra[0-15], rb[0-15] with VOP's mm bits = 0

NOTES:
1. K & S variants to be encoded by a combination of the following:
(a) dividing register bank into signed (r0-7) and unsigned (r8-15) sub-banks
(b) repurposing of VOP "mm" masking bits, to override global Saturation CSR mode bit:
(mm=01 is unused by V extension a could be used to force instruction-by-instruction Saturation)
(c) using VOP's FP/unused opcodes for Halving variants, which only make sense for integer types (eg. VROUND presumably is not needed for integer types?)

2. Alternative vdcfg CSR settings can provide other desired register splits between signed/unsigned and 8/16 bit int

Table 2. SIMD 8-bit Add/Subtract Instructions

Mnemonic	Instruction	Operation
ADD8 rt, ra, rb	8-bit Addition	rt.B*x* = ra.B*x* + rb.B*x*; (*x*=3..0)
RADD8 rt, ra, rb	8-bit Signed Halving Addition	rt.B*x* = (ra.B*x* + rb.B*x*) s>> 1; (*x*=3..0)
URADD8 rt, ra, rb	8-bit Unsigned Halving Addition	rt.B*x* = (ra.B*x* + rb.B*x*) u>> 1; (*x*=3..0)
KADD8 rt, ra, rb	8-bit Signed Saturating Addition	rt.B*x* = SAT.Q7(ra.B*x* + rb.B*x*); (*x*=3..0)
UKADD8 rt, ra, rb	8-bit Unsigned Saturating Addition	rt.B*x* = SAT.U8(ra.B*x* + rb.B*x*); (*x*=3..0)
SUB8 rt, ra, rb	8-bit Subtraction	rt.B*x* = ra.B*x* - rb.B*x*; (*x*=3..0)
RSUB8 rt, ra, rb	8-bit Signed Halving Subtraction	rt.B*x* = (ra.B*x* - rb.B*x*) s>> 1; (*x*=3..0)
URSUB8 rt, ra, rb	8-bit Unsigned Halving Subtraction	rt.B*x* = (ra.B*x* - rb.B*x*) u>> 1; (*x*=3..0)

Mnemonic	Instruction	Operation
KSUB8 rt, ra, rb	8-bit Signed Saturating Subtraction	rt.B*x* = SAT.Q7(ra.B*x* - rb.B*x*); (*x*=3..0)
UKSUB8 rt, ra, rb	8-bit Unsigned Saturating Subtraction	rt.B*x* = SAT.U8(ra.B*x* - rb.B*x*); (*x*=3..0)

2.2.3. 16-bit Shift Instructions

SRA[I]16/SRL[I]16/SLL[I]16 to be mapped to VOP shift instructions in same manner as ADD16/SUB16 ie: map to signed r16-23 and unsigned r24-29 register banks in default vdcfg CSR setting
The "K" (Saturation) and "u" (Rounding) variants could be encoded using VOP's mm field (mm=01 is saturated or rounded shift, mm=00 is standard VOP shift)

Mnemonic	Instruction	Operation
SRA16 rt, ra, rb	16-bit Shift Right Arithmetic	rt.H*x* = ra.H*x* s>> rb[3:0]; (*x*=1..0)
SRAI16 rt, ra, im4u	16-bit Shift Right Arithmetic Immediate	rt.H*x* = ra.H*x* s>> im4u; (*x*=1..0)
SRA16.u rt, ra, rb	16-bit Rounding Shift Right Arithmetic	rt.H*x* = RUND(ra.H*x* s>> rb[3:0]); (*x*=1..0)
SRAI16.u rt, ra, im4u	16-bit Rounding Shift Right Arithmetic Immediate	rt.H*x* = RUND(ra.H*x* s>> im4u); (*x*=1..0)
SRL16 rt, ra, rb	16-bit Shift Right Logical	rt.H*x* = ra.H*x* u>> rb[3:0]; (*x*=1..0)
SRLI16 rt, ra, im4u	16-bit Shift Right Logical Immediate	rt.H*x* = ra.H*x* u>> im4u; (*x*=1..0)
SRL16.u rt, ra, rb	16-bit Rounding Shift Right Logical	rt.H*x* = RUND(ra.H*x* u>> rb[3:0]); (*x*=1..0)
SRLI16.u rt, ra, im4u	16-bit Rounding Shift Right Logical Immediate	rt.H*x* = RUND(ra.H*x* u>> im4u); (*x*=1..0)
SLL16 rt, ra, rb	16-bit Shift Left Logical	rt.H*x* = ra.H*x* << rb[3:0]; (*x*=1..0)

Mnemonic	Instruction	Operation
SLLI16 rt, ra, im4u	16-bit Shift Left Logical Immediate	rt.H*x* = ra.H*x* << im4u; (*x*=1..0)
KSLL16 rt, ra, rb	16-bit Saturating Shift Left Logical	Pseudo instruction of "KSLRA16".
KSLLI16 rt, ra, im4u	16-bit Saturating Shift Left Logical Immediate	rt.H*x* = SAT.Q15(ra.H*x* << im4u); (*x*=1..0)

KSLRA16 rt, ra, rb	16-bit Shift Left Logical with Saturation & Shift Right Arithmetic	if (rb[4:0] < 0)
		rt.H*x* = ra.H*x* s>> -rb[4:0];
		if (rb[4:0] > 0)
		rt.H*x* = SAT.Q15(ra.H*x* << rb[4:0]);
		(*x*=1..0)

KSLRA16.u rt, ra, rb	16-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	if (rb[4:0] < 0)
		rt.H*x* = RUND(ra.H*x* s>> -rb[4:0]);
		if (rb[4:0] > 0)
		rt.H*x* = SAT.Q15(ra.H*x* << rb[4:0]);
		(*x*=1..0)

2.2.4. 16-bit Compare Instructions

CMPEQ16 and [U/S]CMP[LT/LE]16 to be mapped to VOP compare instructions, using signed r16-23 & unsigned r24-29 register banks (in default vdcfg CSR setting)
*** NOTE *** NEED TO HARMONISE CMPLT WITH VOP's VSGE

Table 4. SIMD 16-bit Compare Instructions

Mnemonic	Instruction	Operation
CMPEQ16 rt, ra, rb	16-bit Compare Equal	rt.H*x* = (ra.H*x* == rb.H*x*)? 0xffff : 0; (*x*=1..0)
SCMPLT16 rt, ra, rb	16-bit Signed Compare Less Than	rt.H*x* = (ra.H*x* < rb.H*x*)? 0xffff : 0; (*x*=1..0)
SCMPLE16 rt, ra, rb	16-bit Signed Compare Less Than & Equal	rt.H*x* = (ra.H*x* <= rb.H*x*)? 0xffff : 0; (*x*=1..0)
UCMPLT16 rt, ra, rb	16-bit Unsigned Compare Less Than	rt.H*x* = (ra.H*x* u< rb.H*x*)? 0xffff : 0; (*x*=1..0)

Mnemonic	Instruction	Operation
UCMPLE16 rt, ra, rb	16-bit Unsigned Compare Less Than & Equal	rt.H*x* = (ra.H*x* u<= rb.H*x*)? 0xffff : 0; (*x*=1..0)

2.2.5. 8-bit Compare Instructions

CMPEQ8 and [U/S]CMP[LT/LE]8 to be mapped to VOP compare instructions, using signed r0-7 & unsigned r8-r15 register banks (in default vdcfg CSR setting)
*** NOTE *** NEED TO HARMONISE CMPLT TO VOP's VSGE

Table 5. SIMD 8-bit Compare Instructions

Mnemonic	Instruction	Operation
CMPEQ8 rt, ra, rb	8-bit Compare Equal	rt.B*x* = (ra.B*x* == rb.B*x*)? 0xff : 0; (*x*=3..0)
SCMPLT8 rt, ra, rb	8-bit Signed Compare Less Than	rt.B*x* = (ra.B*x* < rb.B*x*)? 0xff : 0; (*x*=3..0)
SCMPLE8 rt, ra, rb	8-bit Signed Compare Less Than & Equal	rt.B*x* = (ra.B*x* <= rb.B*x*)? 0xff : 0; (*x*=3..0)
UCMPLT8 rt, ra, rb	8-bit Unsigned Compare Less Than	rt.B*x* = (ra.B*x* u< rb.B*x*)? 0xff : 0; (*x*=3..0)
UCMPLE8 rt, ra, rb	8-bit Unsigned Compare Less Than & Equal	rt.B*x* = (ra.B*x* u<= rb.B*x*)? 0xff : 0; (*x*=3..0)

2.2.6. 16-bit Misc Instructions

[S/U]MIN16/MAX16/MUL16 to be mapped to corresponding VOP instructions using r16-29 and mm=00
KMUL16/KABS16/KCLIP16 to be mapped to VOP instructions using r16-29 and mm=01

Table 6. SIMD 16-bit Miscellaneous Instructions

Mnemonic	Instruction	Operation
SMIN16 rt, ra, rb	16-bit Signed Minimum	rt.H*x* = (ra.H*x* < rb.H*x*)? ra.Hx : rb.Hx; (*x*=1..0)
UMIN16 rt, ra, rb	16-bit Unsigned Minimum	rt.H*x* = (ra.H*x* u< rb.H*x*)? ra.Hx : rb.Hx; (*x*=1..0)

Mnemonic	Instruction	Operation
SMAX16 rt, ra, rb	16-bit Signed Maximum	rt.H*x* = (ra.H*x* > rb.H*x*)? ra.Hx : rb.Hx; (*x*=1..0)
UMAX16 rt, ra, rb	16-bit Unsigned Maximum	rt.H*x* = (ra.H*x* u> rb.H*x*)? ra.Hx : rb.Hx; (*x*=1..0)

SCLIP16 rt, ra, imm4u	16-bit Signed Clip Value	n = imm4u;
		rt.H*x* = SAT.Qn(ra.H*x*);
		(*x*=1..0)

UCLIP16 rt, ra, imm4u	16-bit Unsigned Clip Value	m = imm4u;
		rt.H*x* = SAT.Um(ra.H*x*);
		(*x*=1..0)

KHM16 rt, ra, rb	16-bit Signed Multiply	rt.H*x* = SAT.Q15((ra.H*x* s* rb.H*x*) >> 15); (*x*=1..0)
KHMX16 rt, ra, rb	16-bit Crossed Signed Multiply	rt.H*x* = SAT.Q15((ra.H*x* s* rb.H*y*) >> 15); (*x*,*y*)=(1,0), (0,1)
SMUL16 rt, ra, rb	16-bit Signed Multiply to 32-bit	r[tU] = ra.H1 s* rb.H1; r[tL] = ra.H0 s* rb.H0;
SMULX16 rt, ra, rb	16-bit Signed Crossed Multiply to 32-bit	r[tU] = ra.H1 s* rb.H0; r[tL] = ra.H0 s* rb.H1;
UMUL16 rt, ra, rb	16-bit Unsigned Multiply to 32-bit	r[tU] = ra.H1 u* rb.H1; r[tL] = ra.H0 u* rb.H0;
UMULX16 rt, ra, rb	16-bit Unsigned Crossed Multiply to 32-bit	r[tU] = ra.H1 u* rb.H0; r[tL] = ra.H0 u* rb.H1;
KABS16 rt, ra	16-bit Absolute Value	rt.H*x* = SAT.Q15(ABS(ra.H*x*)); (*x*=1..0)

2.2.7. 8-bit Misc Instructions

[S/U]MIN8/MAX8/MUL8 to be mapped to corresponding VOP instructions using r0-r15, and mm=00
KMUL8/KABS8 to be mapped to VOP instructions using r0-15, and mm=01

Table 7. SIMD 8-bit Miscellaneous Instructions

Mnemonic	Instruction	Operation
SMIN8 rt, ra, rb	8-bit Signed Minimum	rt.B*x* = (ra.B*x* < rb.B*x*)? ra.Bx : rb.Bx; (*x*=3..0)
UMIN8 rt, ra, rb	8-bit Unsigned Minimum	rt.B*x* = (ra.B*x* u< rb.B*x*)? ra.Bx : rb.Bx; (*x*=3..0)
SMAX8 rt, ra, rb	8-bit Signed Maximum	rt.B*x* = (ra.B*x* > rb.B*x*)? ra.Bx : rb.Bx; (*x*=3..0)
UMAX8 rt, ra, rb	8-bit Unsigned Maximum	rt.B*x* = (ra.B*x* u> rb.B*x*)? ra.Bx : rb.Bx; (*x*=3..0)
KABS8 rt, ra	8-bit Absolute Value	rt.B*x* = SAT.Q7(ABS(ra.B*x*)); (*x*=3..0)

2.2.8. 8-bit Unpacking Instructions

[S/Z]UNPKD810 mapped to VMV (ie: move from 8bit to 16bit vector type, with auto type promotion)

Table 8. 8-bit Unpacking Instructions

Mnemonic	Instruction	Operation
SUNPKD810 rt, ra	Signed Unpacking Bytes 1 & 0	rt.H*x* = SE16(ra.B*y*); (*x*,*y*) = (1,1), (0,0)
SUNPKD820 rt, ra	Signed Unpacking Bytes 2 & 0	rt.H*x* = SE16(ra.B*y*); (*x*,*y*) = (1,2), (0,0)
SUNPKD830 rt, ra	Signed Unpacking Bytes 3 & 0	rt.H*x* = SE16(ra.B*y*); (*x*,*y*) = (1,3), (0,0)
SUNPKD831 rt, ra	Signed Unpacking Bytes 3 & 1	rt.H*x* = SE16(ra.B*y*); (*x*,*y*) = (1,3), (0,1)
ZUNPKD810 rt, ra	Unsigned Unpacking Bytes 1 & 0	rt.H*x* = ZE16(ra.B*y*); (*x*,*y*) = (1,1), (0,0)

Mnemonic	Instruction	Operation
ZUNPKD820 rt, ra	Unsigned Unpacking Bytes 2 & 0	rt.H*x* = ZE16(ra.B*y*); (*x*,*y*) = (1,2), (0,0)
ZUNPKD830 rt, ra	Unsigned Unpacking Bytes 3 & 0	rt.H*x* = ZE16(ra.B*y*); (*x*,*y*) = (1,3), (0,0)
ZUNPKD831 rt, ra	Unsigned Unpacking Bytes 3 & 1	rt.H*x* = ZE16(ra.B*y*); (*x*,*y*) = (1,3), (0,1)

2.3. Non-SIMD Data Processing Instructions

2.3.1. 32-bit Addition/Subtraction Instructions

There are 4 instructions here.

RADDW & RSUBW equivalent to VADD (with Halving) on vectors of 32 bit elements ie: v30-31 registers

Mnemonic	Instruction	Operation
RADDW rt, ra, rb	32-bit Signed Halving Addition	rt = (ra + rb) s>> 1
URADDW rt, ra, rb	32-bit Unsigned Halving Addition	rt = (ra + rb) u>> 1
RSUBW rt, ra, rb	32-bit Signed Halving Subtraction	rt = (ra - rb) s>> 1
URSUBW rt, ra, rb	32-bit Unsigned Halving Subtraction	rt = (ra - rb) u>> 1

2.3.2. 32-bit Shift Instructions

KSLL[I] equivalent to VSLL[I] on v30-v31 registers, with mm = "01" (Saturation/Rounding on)
SRA[I].u equivalent to VSRA[I] on v30-31 registers, with mm = "01" (Saturation/Rounding on)

Table 10. 32-bit Shift Instructions

Mnemonic	Instruction	Operation
SRA.u rt, ra, rb	Rounding Shift Right Arithmetic	rt = RUND(ra s>> rb[4:0])
SRAI.u rt, ra, imm5u	Rounding Shift Right Arithmetic Immediate	rt = RUND(ra s>> imm5u)
KSLL rt, ra, rb	Saturating Shift Left Logical	rt = SAT.Q31(ra << rb[4:0]); Pseudo instruction of "KSLRAW".
KSLLI rt, ra, imm5u	Saturating Shift Left Logical Immediate	rt = SAT.Q31(ra << imm5u)

KSLRAW.u rt, ra, rb	Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	if (rb[7:0] < 0)
		rt = RUND(ra s>> SAT.U5(-rb[7:0]));
		if (rb[7:0] > 0)
		rt = SAT.Q31(ra << SAT.U5(rb[7:0]));

2.3.3. 16-bit Packing Instructions

There are 4 instructions here.

Table 11. 16-bit Packing Instructions

Mnemonic	Instruction	Operation
PKBB16 rt, ra, rb	Pack two 16-bit data from Bottoms	rt = CONCAT(ra.H0, rb.H0);
PKBT16 rt, ra, rb	Pack two 16-bit data Bottom & Top	rt = CONCAT(ra.H0, rb.H1);
PKTB16 rt, ra, rb	Pack two 16-bit data Top & Bottom	rt = CONCAT(ra.H1, rb.H0);
PKTT16 rt, ra, rb	Pack two 16-bit data from Tops	rt = CONCAT(ra.H1, rb.H1);

2.3.4. Most Significant Word "32x32" Multiply & Add Instructions

SMMUL[.u] equivalent to VMULH on registers r30-31, with mm=00 or mm=01 respectively
Alternatively all four instructions below equivalent to VMUL/VMADD with a 32 bit fixed point fraction type

Table 12. Signed MSW 32x32 Multiply and Add Instructions

Mnemonic	Instruction	Operation
SMMUL rt, ra, rb	MSW "32 x 32" Signed Multiplication (MSW 32 = 32x32)	rt = (ra*rb)[63:32];
SMMUL.u rt, ra, rb	MSW "32 x 32" Signed Multiplication with Rounding (MSW 32 = 32x32)	rt = RUND(ra*rb)[63:32];
KMMAC rt, ra, rb	MSW "32 x 32" Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x32)	rt = SAT.Q31(rt + (ra*rb)[63:32]);
KMMAC.u rt, ra, rb	MSW "32 x 32" Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x32)	rt = SAT.Q31(rt + RUND(ra*rb)[63:32]);

Mnemonic	Instruction	Operation
KMMSB rt, ra, rb	MSW "32 x 32" Signed Multiplication and Saturating Subtraction (MSW 32 = 32 - 32x32)	rt = SAT.Q31(rt - (ra*rb)[63:32]);
KMMSB.u rt, ra, rb	MSW "32 x 32" Signed Multiplication and Saturating Subtraction with Rounding (MSW 32 = 32 - 32x32)	rt = SAT.Q31(rt - RUND(ra*rb)[63:32]);
KWMMUL rt, ra, rb	MSW "32 x 32" Signed Multiplication & Double (MSW 32 = 32x32 << 1)	rt = SAT.Q31((ra*rb << 1)[63:32]);
KWMMUL.u rt, ra, rb	MSW "32 x 32" Signed Multiplication & Double with Rounding (MSW 32 = 32x32 << 1)	rt = SAT.Q31(RUND(ra*rb << 1)[63:32]);

2.3.5. Most Significant Word "32x16" Multiply & Add Instructions

SMMWB[.u] and KMMAWB[.u] are potentially equivalent to VMUL/VMADD where one source register holds a 16 bit fixed point fraction type and destination & other source register are 32 fixed point fraction type

Table 13. Signed MSW 32x16 Multiply and Add Instructions

Mnemonic	Instruction	Operation
SMMWB rt, ra, rb	MSW "32 x Bottom 16" Signed Multiplication (MSW 32 = 32x16)	rt = (ra*rb.L)[47:16];
SMMWB.u rt, ra, rb	MSW "32 x Bottom 16" Signed Multiplication with Rounding (MSW 32 = 32x16)	rt = RUND(ra*rb.L)[47:16];
SMMWT rt, ra, rb	MSW "32 x Top 16" Signed Multiplication (MSW 32 = 32x16)	rt = (ra*rb.H)[47:16];

Mnemonic	Instruction	Operation
SMMWT.u rt, ra, rb	MSW "32 x Top 16" Signed Multiplication with Rounding (MSW 32 = 32x16)	rt = RUND(ra*rb.H)[47:16];
KMMAWB rt, ra, rb	MSW "32 x Bottom 16" Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x16)	rt = SAT.Q31(rt + (ra*rb.L)[47:16]);
KMMAWB.u rt, ra, rb	MSW "32 x Bottom 16" Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	rt = SAT.Q31(rt + RUND(ra*rb.L)[47:16]);
KMMAWT rt, ra, rb	MSW "32 x Top 16" Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x16)	rt = SAT.Q31(rt + (ra*rb.H)[47:16]);
KMMAWT.u rt, ra, rb	MSW "32 x Top 16" Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	rt = SAT.Q31(rt + RUND(ra*rb.H)[47:16]);

2.3.6. Signed 16-bit Multiply with 32-bit Add/Subtract Instructions

SMBB/KMDA are equivalent to VMUL/VMADD, with one source register being a vector of 16 bit integers (r16-29) and the destination register & other source register are 32 bit integers (r30-r31)

KMABB equivalent to VMADD with one source a vector 16 bit integers and other source a 32 bit integer

Mnemonic	Instruction	Operation
SMBB rt, ra, rb	Signed Multiply Bottom 16 & Bottom 16 (32 = 16x16)	rt = ra.L*rb.L;
SMBT rt, ra, rb	Signed Multiply Bottom 16 & Top 16 (32 = 16x16)	rt = ra.L*rb.H;

Mnemonic	Instruction	Operation
SMTT rt, ra, rb	Signed Multiply Top 16 & Top 16 (32 = 16x16)	rt = ra.H*rb.H;
KMDA rt, ra, rb	Two “16x16” and Signed Addition (32 = 16x16 + 16x16)	rt = SAT.Q31(ra.H*rb.H + ra.L*rb.L);
KMXDA rt, ra, rb	Two Crossed “16x16” and Signed Addition (32 = 16x16 + 16x16)	rt = SAT.Q31(ra.H*rb.L + ra.L*rb.H);
SMDS rt, ra, rb	Two “16x16” and Signed Subtraction (32 = 16x16 - 16x16)	rt = (ra.H*rb.H) - (ra.L*rb.L);
SMDRS rt, ra, rb	Two “16x16” and Signed Reversed Subtraction (32 = 16x16 - 16x16)	rt = (ra.L*rb.L) - (ra.H*rb.H);
SMXDS rt, ra, rb	Two Crossed “16x16” and Signed Subtraction (32 = 16x16 - 16x16)	rt = (ra.H*rb.L) - (ra.L*rb.H);
KMABB rt, ra, rb	“Bottom 16 x Bottom 16” with 32-bit Signed Addition (32 = 32 + 16x16)	rt = SAT.Q31(rt + ra.L*rb.L);
KMABT rt, ra, rb	“Bottom 16 x Top 16” with 32-bit Signed Addition (32 = 32 + 16x16)	rt = SAT.Q31(rt + ra.L*rb.H);
KMATT rt, ra, rb	“Top 16 x Top 16” with 32-bit Signed Addition (32 = 32 + 16x16)	rt = SAT.Q31(rt + ra.H*rb.H);
KMADA rt, ra, rb	Two “16x16” with 32-bit Signed Double Addition (32 = 32 + 16x16 + 16x16)	rt = SAT.Q31(rt + ra.H*rb.H + ra.L*rb.L);
KMAXDA rt, ra, rb	Two Crossed “16x16” with 32-bit Signed Double Addition (32 = 32 + 16x16 + 16x16)	rt = SAT.Q31(rt + ra.H*rb.L + ra.L*rb.H);

Mnemonic	Instruction	Operation
KMADS rt, ra, rb	Two "16x16" with 32-bit Signed Addition and Subtraction (32 = 32 + 16x16 - 16x16)	rt = SAT.Q31(rt + ra.H*rb.H - ra.L*rb.L);
KMADRS rt, ra, rb	Two "16x16" with 32-bit Signed Addition and Reversed Subtraction (32 = 32 + 16x16 - 16x16)	rt = SAT.Q31(rt + ra.L*rb.L - ra.H*rb.H);
KMAXDS rt, ra, rb	Two Crossed "16x16" with 32-bit Signed Addition and Subtraction (32 = 32 + 16x16 - 16x16)	rt = SAT.Q31(rt + ra.H*rb.L - ra.L*rb.H);
KMSDA rt, ra, rb	Two "16x16" with 32-bit Signed Double Subtraction (32 = 32 - 16x16 - 16x16)	rt = SAT.Q31(rt - ra.H*rb.H - ra.L*rb.L);
KMSXDA rt, ra, rb	Two Crossed "16x16" with 32-bit Signed Double Subtraction (32 = 32 - 16x16 - 16x16)	rt = SAT.Q31(rt - ra.H*rb.L - ra.L*rb.H);

2.3.7. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions

Table 15. Signed 16-bit Multiply 64-bit Add/Subtract Instructions

Mnemonic	Instruction	Operation

SMAL rt, ra, rb	"16 x 16" with 64-bit Signed Addition (64 = 64 + 16x16)	a64 = r[aU].r[aL];
		t64 = a64 + rb.H*rb.L;
		r[tU].r[tL] = t64;

2.3.8. Miscellaneous Instructions

There are 7 instructions here.

Table 16. Miscellaneous Instructions

Mnemonic	Instruction	Operation
SCLIP32 rt, ra, imm5u	Signed Clip Value	n = imm5u; rt = SAT.Qn(ra);
UCLIP32 rt, ra, imm5u	Unsigned Clip Value	m = imm5u; rt = SAT.Um(ra);
BITREV rt, ra, rb	Bit Reverse	w = rb[4:0]; rt = ra[0:31] u>> (31- w);
BITREVI rt, ra, imm5u	Bit Reverse Immediate	w = imm5u; rt = ra[0:31] u>> (31- w);
WEXT rt, ra, rb	Extract 32-bit from a 64-bit value	a64 = r[aU].r[aL]; lsb = rb[4:0] rt = a64[(31+lsb):lsb];
WEXTI rt, ra, imm5u	Extract 32-bit from a 64-bit value Immediate	a64 = r[aU].r[aL]; lsb = imm5u rt = a64[(31+lsb):lsb];
BPICK rt, ra, rb, rc	Bit-wise Pick	rt[i] = rc[i]? ra[i] : rb[i]; (i=31..0)
INSB rt, ra, imm2u	Insert Byte	insLSB = imm2u*8; rt[(insLSB+7):insLSB] = ra[7:0];
KABS Rd, Rs1	Absolute with Saturation	If (Rs1 == 0x80000000) { Rd=0x7fffffff; OV=1; } else { Rd = | Rs1 |; }
UCLIP32 Rd, Rs1, imm5u	Clip Value	If (Rs1 > 2^{imm5u} -1) { Rd=2^{imm5u} -1; OV=1; } else if (Rs1 < 0) { Rd=0; OV=1; } } else { Rd=Rs1; }
SCLIP32 Rd, Rs1, imm5u	Clip Value Signed	If (Rs1 > 2^{imm5u} -1) { Rd=2^{imm5u} -1; OV=1;

Mnemonic	Instruction	Operation
		} else if (Rs1 < -2 ^{imm5u}) { Rd=-2 ^{imm5u}; OV=1; } else { Rd=Rs1; }
CLZ Rd, Rs1	Count leading zero	Rd = COUNT_ZERO_FROM_MSB(Rs1)
CLO Rd, Rs1	Count leading one	Rd = COUNT_ONE_FROM_MSB(Rs1)
MAX Rd, Rs1, Rs2	Return the larger signed value	Rd = signed-max(Rs1, Rs2)
MIN Rd, Rs1, Rs2	Return the smaller signed value	Rd = signed-min (Rs1, Rs2)
AVE Rd, Rs1, Rs1	Average two signed integers with rounding	Rd = (Rs1 + Rs2 + 1) (arith) >> 1
PBSAD Rd, Rs1, Rs1	Parallel Byte Sum of Absolute Difference	a = ABS(Rs1(7,0) – Rs2(7,0)); b = ABS(Rs1(15,8) – Rs2(15,8)); c = ABS(Rs1(23,16) – Rs2(23,16)); d = ABS(Rs1(31,24) – Rs2(31,24)); Rd = a + b + c + d;
PBSADA Rd, Rs1, Rs1	Parallel Byte Sum of Absolute Difference Accumulate	a = ABS(Rs1(7,0) – Rs2(7,0)); b = ABS(Rs1(15,8) – Rs2(15,8)); c = ABS(Rs1(23,16) – Rs2(23,16)); d = ABS(Rs1(31,24) – Rs2(31,24)); Rd = Rd + a + b + c + d;

2.3.9. Q31 saturation Instructions

KADDW, KSUBW, KDMBB are equivalent to VADD/VSUB/VMUL where both source registers hold vector of 16 bit integer elements (v16-29) and destination register is 32 bit integer (v30-31), and mm=01

Table 17. Q31 saturation ALU Instructions

Mnemonic	Instruction	Operation
KADDW Rt, Ra, Rb	Add with Q31 saturation.	Rt = SAT.Q31(Ra + Rb)
KSUBW Rt, Ra, Rb	Subtract with Q31 saturation.	Rt = SAT.Q31(Ra – Rb)

Mnemonic	Instruction	Operation
KSLRAW Rt, Ra, Rb	Logical left shift or arithmetic right shift with Q31 saturation.	(Rb[7:0] >=0) ? Rt = SAT.Q31(Ra << Rb[7:0]): Rt = (Ra s>> -Rb[7:0])
KDMBB Rt, Ra, Rb	Multiply Q15 numbers with Q31 saturation in bottom parts of two registers.	Rt = SAT.Q31(Ra.H0 * Rb.H0)
KDMTB Rt, Ra, Rb	Multiply Q15 numbers with Q31 saturation in top and bottom parts of two registers.	Rt = SAT.Q31(Ra.H1 * Rb.H0)
KDMBT Rt, Ra, Rb	Multiply Q15 numbers with Q31 saturation in bottom and top parts of two registers.	Rt = SAT.Q31(Ra.H0 * Rb.H1)
KDMTT Rt, Ra, Rb	Multiply Q15 numbers with Q31 saturation in top parts of two registers.	Rt = SAT.Q31(Ra.H1 * Rb.H1)

2.3.10. Q15 saturation instructions

The following table lists instructions related to Q15 arithmetic.

Table 18. Q15 saturation ALU Instructions

Mnemonic	Instruction	Operation
KADDH Rt, Ra, Rb	Add with Q15 saturation.	Rt = SAT.Q15(Ra + Rb)
KSUBH Rt, Ra, Rb	Subtract with Q15 saturation	Rt = SAT.Q15(Ra – Rb)
KHMBB Rt, Ra, Rb	Multiply Q15 numbers in bottom parts of two registers and extract high part with Q15 saturation.	Rt = SAT.Q15((Ra.H0 * Rb.H0) s>> 15)
KHMTB Rt, Ra, Rb	Multiply Q15 numbers in top and bottom parts of two registers and extract high part with Q15 saturation.	Rt = SAT.Q15((Ra.H1 * Rb.H0) s>> 15)
KHMBT Rt, Ra, Rb	Multiply Q15 numbers in bottom and top parts of two registers and extract	Rt = SAT.Q15((Ra.H0 * Rb.H1) s>> 15)

	high part with Q15 saturation.	
KHMTT Rt, Ra, Rb	Multiply Q15 numbers in top parts of two registers and extract high part with Q15 saturation.	Rt = SAT.Q15((Ra.H1 * Rb.H1) s>> 15)

2.3.11. Overflow status manipulation instructions

The following table lists the user instructions related to Overflow (OV) flag manipulation.

Table 19. OV (Overflow) flag Set/Clear Instructions

Mnemonic	Instruction	Operation
RDOV Rt	Read mxstatus.OV to Rt.	Rt = ZE32(mxstatus.OV)
CLROV	Clear mxstatus.OV flag.	mxstatus.OV = 0

2.4. 64-bit Instructions

2.4.1. 64-bit Addition & Subtraction Instructions

Table 20. 64-bit Add/Subtract Instructions

Mnemonic	Instruction	Operation

ADD64 rt, ra, rb	64-bit Addition	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = a64 + b64;
		r[tU].r[tL] = t64;

RADD64 rt, ra, rb	64-bit Signed Halving Addition	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = (a64 + b64) s>>1;
		r[tU].r[tL] = t64;

URADD64 rt, ra, rb	64-bit Unsigned Halving Addition	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = (a64 + b64) u>>1;
		r[tU].r[tL] = t64;

KADD64 rt, ra, rb	64-bit Signed Saturating Addition	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = SAT.Q63(a64 + b64);
		r[tU].r[tL] = t64;

UKADD64 rt, ra, rb	64-bit Unsigned Saturating Addition	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = SAT.U64(a64 + b64);
		r[tU].r[tL] = t64;

Mnemonic	Instruction	Operation

SUB64 rt, ra, rb	64-bit Subtraction	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = a64 - b64;
		r[tU].r[tL] = t64;

RSUB64 rt, ra, rb	64-bit Signed Halving Subtraction	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = (a64 - b64) s>>1;
		r[tU].r[tL] = t64;

URSUB64 rt, ra, rb	64-bit Unsigned Halving Subtraction	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = (a64 - b64) u>>1;
		r[tU].r[tL] = t64;

KSUB64 rt, ra, rb	64-bit Signed Saturating Subtraction	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = SAT.Q63(a64 - b64);
		r[tU].r[tL] = t64;

UKSUB64 rt, ra, rb	64-bit Unsigned Saturating Subtraction	a64 = r[aU].r[aL]; b64 = r[bU].r[bL];
		t64 = SAT.U64(a64 - b64);
		r[tU].r[tL] = t64;

2.4.2. 32-bit Multiply with 64-bit Add/Subtract Instructions

Table 21. 32-bit Multiply 64-bit Add/Subtract Instructions

Mnemonic	Instruction	Operation

SMAR64 rt, ra, rb	32x32 with 64-bit Signed Addition	c64 = r[tU].r[tL];
		t64 = c64 + ra*rb; // signed
		r[tU].r[tL] = t64;

SMSR64 rt, ra, rb	32x32 with 64-bit Signed Subtraction	c64 = r[tU].r[tL];
		t64 = c64 - ra*rb; // signed
		r[tU].r[tL] = t64;

UMAR64 rt, ra, rb	32x32 with 64-bit Unsigned Addition	c64 = r[tU].r[tL];
		t64 = c64 + ra*rb; // unsigned
		r[tU].r[tL] = t64;

UMSR64 rt, ra, rb	32x32 with 64-bit Unsigned Subtraction	c64 = r[tU].r[tL];
		t64 = c64 - ra*rb; // unsigned
		r[tU].r[tL] = t64;

KMAR64 rt, ra, rb	32x32 with Saturating 64-bit Signed Addition	c64 = r[tU].r[tL];
		t64 = SAT.Q63(c64 + ra*rb);
		r[tU].r[tL] = t64;

KMSR64 rt, ra, rb	32x32 with Saturating 64-bit Signed Subtraction	c64 = r[tU].r[tL]; t64 = SAT.Q63(c64 – ra*rb);

Mnemonic	Instruction	Operation
		r[tU].r[tL] = t64;

UKMAR64 rt, ra, rb	32x32 with Saturating 64-bit Unsigned Addition	c64 = r[tU].r[tL];
		t64 = SAT.U64(c64 + ra*rb);
		r[tU].r[tL] = t64;

UKMSR64 rt, ra, rb	32x32 with Saturating 64-bit Unsigned Subtraction	c64 = r[tU].r[tL];
		t64 = SAT.U64(c64 - ra*rb);
		r[tU].r[tL] = t64;

2.4.3. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions

Table 22. Signed 16-bit Multiply 64-bit Add/Subtract Instructions

Mnemonic	Instruction	Operation

SMALBB rt, ra, rb	"Bottom 16 x Bottom 16" with 64-bit Signed Addition (64 = 64 + 16x16)	c64 = r[tU].r[tL];
		t64 = c64 + ra.L*rb.L;
		r[tU].r[tL] = t64;

SMALBT rt, ra, rb	"Bottom 16 x Top 16" with 64-bit Signed Addition (64 = 64 + 16x16)	c64 = r[tU].r[tL];
		t64 = c64 + ra.L*rb.H;
		r[tU].r[tL] = t64;

SMALTT rt, ra, rb	"Top 16 x Top 16" with 64-bit Signed Addition (64 = 64 + 16x16)	c64 = r[tU].r[tL]; t64 = c64 + ra.H*rb.H;

Mnemonic	Instruction	Operation
		r[tU].r[tL] = t64;

SMALDA rt, ra, rb	Two “16x16” with 64-bit Signed Double Addition
	(64 = 64 + 16x16 + 16x16)
		c64 = r[tU].r[tL];
		t64 = c64 + ra.H*rb.H + ra.L*rb.L;
		r[tU].r[tL] = t64;

SMALXDA rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Double Addition
	(64 = 64 + 16x16 + 16x16)
		c64 = r[tU].r[tL];
		t64 = c64 + ra.H*rb.L + ra.L*rb.H;
		r[tU].r[tL] = t64;

SMALDS rt, ra, rb	Two “16x16” with 64-bit Signed Addition and Subtraction
	(64 = 64 + 16x16 - 16x16)
		c64 = r[tU].r[tL];
		t64 = c64 + ra.H*rb.H - ra.L*rb.L;
		r[tU].r[tL] = t64;

SMALDRS rt, ra, rb	Two “16x16” with 64-bit Signed Addition and Reversed Subtraction
	(64 = 64 + 16x16 - 16x16)
		c64 = r[tU].r[tL];
		t64 = c64 + ra.L*rb.L - ra.H*rb.H;
		r[tU].r[tL] = t64;

SMALXDS rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Addition and Subtraction
	(64 = 64 + 16x16 - 16x16)
		c64 = r[tU].r[tL];
		t64 = c64 + ra.H*rb.L - ra.L*rb.H;
		r[tU].r[tL] = t64;

SMSLDA rt, ra, rb	Two “16x16” with 64-bit Signed Double Subtraction (64 = 64 - 16x16 - 16x16)	c64 = r[tU].r[tL]; t64 = c64 - ra.H*rb.H - ra.L*rb.L;

Mnemonic	Instruction	Operation
		r[tU].r[tL] = t64;

SMSLXDA rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Double Subtraction (64 = 64 - 16x16 - 16x16)	c64 = r[tU].r[tL];
		t64 = c64 - ra.H*rb.L - ra.L*rb.H;
		r[tU].r[tL] = t64;

2.5. Zero-Overhead Loop (ZOL) Mechanism Instructions

The following table lists the instructions in the Zero-Overhead Loop Mechanism.

Table 23. ZOL Mechanism Instructions

Mnemonic	Instruction	Operation
MTLBI imm16s	Move to Loop Begin register Immediate.	LB = PC + SE32(imm16s << 1)
MTLEI imm16s	Move to Loop End register Immediate.	LE = PC + SE32(imm16s << 1)

3. User-mode CSR Registers

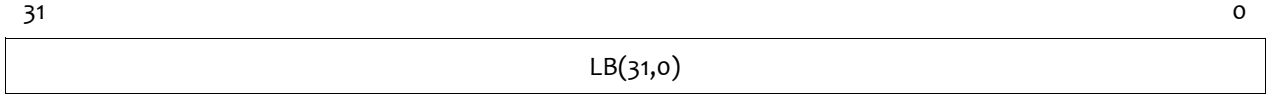
3.1. Loop Begin Register

Mnemonic Name: LB

Presence Condition: TBD

CSR Encoding: TBD

This register stores the beginning address of a loop. It is used by the Zero-Overhead Loop mechanism. If the value of this register is larger than the value of the Loop End (LE) register while the zero-overhead loop mechanism is turned on, UNPREDICTABLE behavior may happen. It is a 32-bit register. And its format is as follows:



Bit 0 will be ignored by hardware. Setting it to 1 will not generate an “unaligned” exception.

It is a RW type register. Its reset value is defined as DC (Don’t Care).

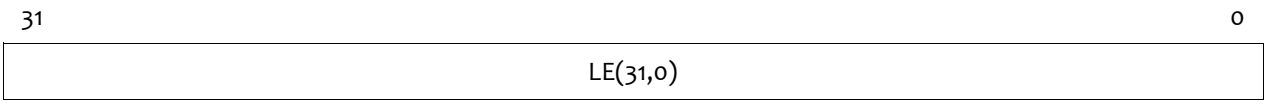
3.2. Loop End Register

Mnemonic Name: LE

Presence Condition: TBD

CSR Encoding: TBD

This register stores the ending address of a loop. It is used by the Zero-Overhead Loop mechanism. If the value of this register is smaller than the value of the Loop End (LE) register while the zero-overhead loop mechanism is turned on, UNPREDICTABLE behavior may happen. It is a 32-bit register. And its format is as follows:



Bit 0 will be ignored by hardware when hardware logic compares the value of program counter with the value of this register.

It is a RW type register. Its reset value is defined as DC (Don't Care).

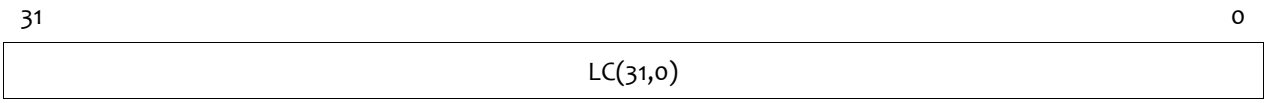
3.3. Loop Count Register

Mnemonic Name: LC

Presence Condition: TBD

CSR Encoding: TBD

This register contains the loop count number that the Zero-Overhead looping operation will be performed. The Zero-Overhead loop mechanism will be turned on when the value of LC is greater than 1. It is a 32-bit register. And its format is as follows:



It is a RW type register. Its reset value is defined as 0.