System for RISC-V P Extension Instruction Names

John Hauser

January 26, 2024

Warning! This document is currently only a draft. The naming scheme described here is liable to change before being accepted by the RISC-V International Association.

This document specifies a system of general rules for the construction of names of RISC-V instructions added by the P extension.

In this naming system, the rules differ somewhat for instructions that perform *scalar* functions versus those that are *packed-SIMD*. The first rule therefore concerns the categorization of instructions as either *scalar* or *packed-SIMD*:

Packed-SIMD versus scalar instructions: An instruction is *packed-SIMD* only if at least one source operand is interpreted as an array of two or more data elemnts. When an instruction's operands are all interpreted as single data elements, the instruction is necessarily classified as *scalar*.

After the introduction of some common rules for operation names, the subsequent section explains the system for naming packed-SIMD instructions.

While there is value in instruction names being compact, the naming system here values readability over brevity. Typing time, measured by number of characters, is not considered the most important factor, especially for the more specialized instructions that are not likely to appear often in program sources. Besides aiding comprehension, a more regular system for instruction names may also ease the addition of new instructions in possible future extensions.

1 Elemental operation names

The functions of many instructions include common elemental operations such as addition and multiplication. Table 1 lists names used for some of these elemental operations. In the simple cases where an instruction performs just a single elemental operation, the full instruction name may be just the elemental name, like ADD or CLZ. More complex instructions may be defined as combinations of elemental operations, with instruction names that contain within them one or more elemental names.

As noted in the table, some of these elemental names (AADD, CLZ, etc.) have precedents in existing ratified RISC-V extensions.

The three-letter mnemonic 'CLS' was chosen for count leading redundant sign bits *instead of 'CLRS' to better match the* `countls` *functions of technical report ISO/IEC TR 18037,* Programming languages: C: Extensions to support embedded processors.

Unlike the base RISC-V comparison instructions SEQ, SLT, etc., which have a result value of either 1 (true) or 0 (false), the *mask-set* operations of Table 1 are defined to set all bits of the result the same, either all ones for true, or all zeros for false.

The RISC-V V extension has "mask set" instructions with names such as VMSEQ.VV (set if equal), VMSLT.VV (set if less than), and VMSNE.VV (set if not equal). For each elemental comparison, these instructions set an individual bit of a mask vector *destination register to 1 for true or 0 for false. These instructions can be said to set "all result bits the same" for each elemental comparison, taking into account that each elemental result is only a single bit. However, while this is true enough, the affinity of the mask-set operations of Table 1 with these V-extension instructions is really more in the purpose of the comparisons, to construct a mask vector for subsequent instructions (vector or packed-SIMD).*

For the "high multiplication" operation, MULH, if the widths of the two arguments are w and x bits respectively, with $w \geq x$, then the function result is defined to be the uppermost w bits of the full $(w+x)$-bit product of the arguments. When the two arguments have equal width, $w = x$, this function is the same as computed by instruction MULH of the M extension, i.e., the uppermost w bits of the full $2w$-bit product of the arguments (which can also be described as the full double-width product shifted right by w bits).

The "Q-format" multiplication, MULQ, is similar to MULH, but is applicable only when the two arguments have the same width and when both arguments are interpreted as signed values, not unsigned. If both arguments are w bits wide, then MULQ computes the full double-width product shifted right by $w - 1$ bits. If this result would be corrupted by overflow (possible only when multiplying two minimum negative values $-2^{w-1} \times -2^{w-1}$), then the result is usually saturated implicitly to the maximum positive value, $2^{w-1} - 1$. (In contrast, MULH and MULHR can never overflow.)

Name	Precedent	Function
AADD	V	averaging addition, $(a+b)/2$
ABS	*common*	absolute value (giving unsigned result, not saturated)
ADD	*common*	addition
ASUB	V	averaging subtraction, $(a-b)/2$
CLS		count leading redundant sign bits
CLZ	Zbb	count leading zero bits
DIF		absolute difference, $\|a-b\|$
MAX	*common*	maximum
MIN	*common*	minimum
MSEQ	V	mask-set if equal (result value is all 1s or all 0s)
MSLT	V	mask-set if less than "
MUL	*common*	multiplication (discarding upper bits as necessary)
MULH	M	high multiplication (discarding lower bits as necessary)
MULHR		MULH with rounding
MULQ		"Q-format" multiplication, $(a \times b)$>>$(width-1)$
MULQR		MULQ with rounding
SABS		saturating absolute value
SADD	V	saturating addition
SAT		saturate to specified bit-width
SHA		shift left or right, arithmetic (signed)
SHAR		SHA with rounding
SLL	I	shift left, logical
SRA	I	shift right, arithmetic
SRAR		SRA with rounding
SRL	I	shift right, logical
SSHA		saturating SHA
SSHAR		saturating SHA with rounding
SSLA		saturating shift left, arithmetic (signed)
SSUB	V	saturating subtraction
SUB	*common*	subtraction
WADD	V	widening addition (double-width result)
WMUL	V	widening multiplication "
WSLA		widening shift left, arithmetic (signed) "
WSLL		widening shift left, logical "
WSUB	V	widening subtraction "
NCLIP	V	narrowing shift right and saturate (double-width input)
NCLIPR		NCLIP with rounding "
NSRA	V	narrowing shift right, arithmetic "
NSRAR		NSRA with rounding "
NSRL	V	narrowing shift right, logical "

Table 1: Names for some elemental operations, in alphabetical order.

2 Combining and augmenting operation names

Many P-extension instructions perform a chained sequence of operations, the most common being versions of *multiply-add*, a multiplication followed by an addition or subtraction. A general rule applies to the names of such instructions:

Operation chains: When an instruction performs a chained sequence of operations, the instruction name usually contains names for the individual operations in left-to-right, first-to-last order: ⟨*op1*⟩ ⟨*op2*⟩ ....

Some examples of scalar (non-SIMD) instructions that perform chained operations are:

MHRACC = MHR (high multiplication with rounding, first op) + ACC (accumulate, second op)
WADDA = WADD (widening addition, first op) + A (accumulate, second op)

The names used for the component operations in a chain may be taken verbatim from Table 1, or they may be alternative or abbreviated names following additional rules. In the two examples above, only one of the four component names is directly from Table 1 ('WADD', but not MHR', 'ACC', or 'A'). The other three are explained below.

The abbreviation 'MHR' in place of 'MULHR' comes from this rule:

Abbreviation of MUL in chains: When a chain of operations includes multiplication, 'MUL' is usually abbreviated as just 'M', so MUL, MULH, MULHR, MULQ, MULQR, or WMUL is denoted by 'M', 'MH', 'MHR', 'MQ', 'MQR', or 'WM', respectively.

The rule for abbreviating 'MUL' is consistent with existing practice for the F and V extensions, which define instructions such as FMADD.S (floating-point multiply-add, single-precision) and VMACC.VV (vector multiply-accumulate, vector-vector), with the multiplication being represented by just the letter 'M'.

An *accumulate* operation in a chain is defined as an addition of a computed value with the previous value of the instruction's destination operand. An accumulate operation is always the last in a chain. If the destination operand is twice as wide as the value being added to it, then the operation is called a *widening accumulate*, similar to a widening addition. The following rule applies to instructions that accumulate:

Accumulate operations: In an instruction name, an accumulate operation is usually abbreviated simply by the letter 'A', or by 'WA' for a widening accumulate. However, if the operation chain consists only of a multiplication and an accumulate (two operations total), then the multiplication gets abbreviated by its usual rule, and the accumulate is denoted in longer form as 'ACC' or 'WACC'.

Thus, the combination of widening addition + accumulate is 'WADDA', but multiplication + accumulate is 'MACC', not 'MA' or 'MULA'.

Keeping with existing RISC-V convention, unsigned operands are usually indicated by the letter 'U':

Signedness of operands/operations: When the signedness of values (*signed* versus *unsigned*) matters, then an instruction's operands are normally interpreted as signed by default. If operands should instead be considered unsigned, and this fact is not indicated in some other way, then the

instruction name may have the letter 'U' suffixed to the operation chain: ⟨*op-chain*⟩ U. If the first source operand should be interpreted as *signed* and the second operand *unsigned*, then the instruction name may have 'SU' suffixed to the operation chain: ⟨*op-chain*⟩ SU.

Suffixing 'U' has many obvious precedents for RISC-V: SLTU, MULHU, DIVU, etc. The 'SU' suffix applies most often to multiplications, as in instruction MULHSU of the M extension.

3 Conventions for packed-SIMD instruction names

3.1 Basic packed-SIMD instructions

The most basic packed-SIMD instructions are those that subdivide the bits of their register operands unambiguously into equal-sized lanes, and compute each lane's result entirely independently of the other lanes, all lanes in parallel. The names of instructions of this kind usually follow a common form:

Basic packed-SIMD instructions: If a packed-SIMD instruction has operands that are all single registers, and if the instruction's computation has the simple structure that each *lane* (whether bytes, halfwords, or words) is entirely self-contained and not dependent on values from other lanes, then the instruction name usually takes the regular form

P ⟨*operations*⟩ . ⟨*S*⟩

where ⟨*operations*⟩ indicates the per-lane computation and ⟨*S*⟩ is a letter for the element width, 'B' for byte, 'H' for halfword, or 'W' for word.

Some example names for basic packed-SIMD instructions are the following:

PADD.B	=	P ADD .B
PMULQ.H	=	P MULQ .H
PMULHSU.H	=	P MULH SU .H
PMAXU.B	=	P MAX U .B
PMHRACC.W	=	P {MulHR + ACC} .W

In each case, the specified computation (ADD, MULQ, etc.) is performed separately within each lane, with lanes being bytes ('.B'), halfwords ('.H'), or words ('.W').

Note that instructions of this form with word elements ('.W') are possible only for RV64. Given that each RV32 register can hold only one 32-bit word, word-width computations with single-register operands are necessarily scalar instructions for RV32, not packed-SIMD.

3.2 Widening and narrowing packed-SIMD instructions

Some packed-SIMD instructions have double-width operands held in even-odd pairs of registers. For *widening* packed-SIMD instructions, the source operands are all single registers (or immediates), and only the destination is a register pair.

Widening packed-SIMD instructions: If the name of a packed-SIMD instruction includes an explicit widening operation such as WADD, WSUB, WMUL, or WACC, then usually the source operands are single registers (or an instruction immediate) and the destination operand is implicitly an even-odd register pair.

The following are some examples of widening packed-SIMD instructions:

PWADD.B	=	P WADD .B
PWSUBAU.H	=	P {WSUB + Accum} U .H
PWMULSU.B	=	P WMUL SU .B
PWMACC.H	=	P {WMul + ACC} .H
PMQRWACC.H	=	P {MulQR + WACC} .H

For these instructions, the appearance of the widening operation in the name (WADD, WSUB, WMUL, WACC, or an abbreviation) is the indication that the destination is a register pair.

There are also *narrowing* packed-SIMD instructions, where the first source operand is a register pair and the other operands are single registers (or immediates).

Narrowing packed-SIMD instructions: If the name of a packed-SIMD instruction includes an explicit narrowing operation such as NSRL (narrowing shift right, logical) or NCLIP, then usually the first source operand is implicitly an even-odd register pair, and the other source operand and destination are single registers (or an instruction immediate).

These are examples of narrowing packed-SIMD instructions:

PNSRL.H	=	P NSRL .H
PNCLIPRU.B	=	P NCLIPR U .B
PNSRAR.H	=	P NSRAR .H

As with widening instructions, the appearance of the narrowing operation in the name (NSRL, NCLIPR, NSRAR) is the indication that the first source operand is a register pair.

3.3 Double-wide packed-SIMD instructions

In addition to widening and narrowing instructions, other double-wide packed-SIMD instructions take even-odd register pairs for both source and destination operands.

Double-wide packed-SIMD instructions: For RV32, when all register operands of a packed-SIMD instruction are even-odd register pairs (64 bits, sources and destination), then the instruction's suffix for element width has the form '. D $\langle S \rangle$' instead of just '. $\langle S \rangle$', the 'D' standing for *double-word* and $\langle S \rangle$ being one of 'B', 'H', or 'W' as usual.

Some examples of RV32 double-wide packed-SIMD instructions are:

PADD.DH = P ADD .DH
PMSEQ.DB = P MSEQ .DB
PASUBU.DW = P ASUB U .DW

The same computation that PADD.H does in RV32 for two halfword lanes (the operands being single 32-bit registers), PADD.DH does over four halfword lanes (64-bit register pairs). PADD.DH is thus equivalent to two PADD.H instructions, one for the even-numbered registers and the other for the odd-numbered registers. In the same way, PMSEQ.DB is equivalent to two PMSEQ.B instructions, while PASUBU.DW is equivalent to two scalar ASUBU instructions.

For consistency with '.DB', '.DH', and '.DW', one might expect the names of regular RV32 packed-SIMD instructions with single-register operands to have suffixes '.WB' and '.WH', with 'W' for word. *However, 32-bit word width is usually implicit for RV32, not explicitly indicated by the letter 'W'. Consider for example the basic addition instruction is ADD, not 'ADDW'; etc. Likewise, the names of RV64 packed-SIMD instructions with single-register operands do not have suffixes '.DB', '.DH', or '.DW', because doubleword width is usually implicit for RV64 instructions.*

3.4 Other mixed-width packed-SIMD instructions

Mixed-width packed-SIMD instructions: If a packed-SIMD instruction has operands that are single registers, and if the elements taken from one or both source operands are half the widths of the computed result elements (byte sources with halfword results, or halfword sources with word results), then the instruction name has two suffixes for element widths, in one of these forms:

$$\text{P}\,\langle operations\rangle\,.\,\langle S\rangle\,.\,\langle T\rangle\langle p\rangle \quad \text{or} \quad \text{P}\,\langle operations\rangle\,.\,\langle S\rangle\,.\,\langle T\rangle\langle p\rangle\langle p\rangle$$

In both cases, $\langle S\rangle$ specifies the *dominant* element width, 'H' for halfword or 'W' for word, while $\langle T\rangle$ is half that size, respectively 'B' for byte or 'H' for halfword. The dominant width is the width of the result elements that are written to the destination.

For the first form of instruction name, ending in only one $\langle p\rangle$, the elements from the instruction's first source operand are the same dominant width as the destination, and the half-width denoted by $\langle T\rangle$ applies only to the second source operand. The final component $\langle p\rangle$ is a letter 'E' or 'O' indicating whether the half-width elements of the second source operand are taken from the even-numbered or odd-numbered positions of the register.

For the second form of instruction name, both source operands supply half-width elements to the instruction, and the two $\langle p\rangle$'s specify the element positions ('E' for even-numbered, 'O' for odd-numbered) within the first and second source registers respectively.

The following are examples of instructions that extract half-width elements from the second source operand, while the first source operand and destination both have the dominant element width:

PMULHSU.H.BE = P MULH SU .H .BE (even)
PMHACC.W.HO = P {MulH + ACC} .W .HO (odd)

The first example performs high signed × unsigned multiplications (MULHSU) of 16-bit halfwords and 8-bit bytes, where the bytes are from the even-numbered positions of the second source operand. The second instruction similarly performs high-multiply-accumulate of words times halfwords, taking the halfwords from the odd-numbered positions of the second source operand.

In the next examples, half-width elements are taken from both source operands, and only the destination has elements of the dominant width:

PMULU.H.BEO = P {widening MUL} U .H .BEO (even & odd)
PMQRACC.W.HEE = P {MulQR + ACC} .W .HEE (even & even)

For instruction PMULU.H.BEO, the multiplications are inherently widening because the source elements are bytes while the dominant element width is hafwords. As a general rule, in this situation, the specified operation (or the first operation in a chain) implicitly widens to the dominant element width.

On the other hand, the multiplications done by PMQRACC.W.HEE are not described above as widening, because a "Q-format" multiplication, MULQR, performs a function different than a widening multiplication. The 'MQR' in the instruction name explicitly indicates that the lower bits of the full 32-bit products will be shifted off, with rounding, as specified in Section 1. To perform the same function without shifting down the products, the correct instruction would be PMACC.W.HEE, with MULQR ('MQR') replaced by MUL ('M', implicitly widening).

Packed-SIMD instructions with both double- and single-wide sources: For RV32, when the first source operand and destination of a packed-SIMD instruction are even-odd register pairs (64 bits) but the second source operand is a single register (32 bits), then the instruction's name has two suffixes as follows:

P ⟨*operations*⟩ .D ⟨S⟩ . ⟨T⟩

The first suffix, '.D ⟨S⟩' specifies that the first source operand and destination are register pairs with dominant element width ⟨S⟩, either 'H' for halfword or 'W' for word; while the second suffix, '. ⟨T⟩', indicates that the second source operand is a single register with half-width elements, ⟨T⟩ being correspondingly 'B' for byte or 'H' for halfword.

These instructions are not common, but the following is an RV32 instruction where the first source operand and destination are register pairs and the second source operand is a single register:

PADD.DW.W = P ADD .DW .W

3.5 Packed-SIMD instructions with horizontal addition/subtraction

A number of packed-SIMD instructions perform *horizontal* additions or subtractions, meaning additions or subtractions across SIMD lanes. A major subcategory of these instructions first peform element-wise multiplications and then add or subtract groups of the products across lanes.

Packed-SIMD instructions with horizontal add/subtract of products: When an instruction performs packed-SIMD element-wise multiplications and then adds groups of products horizontally (across lanes), these additions are indicated in the instruction name by '⟨n⟩ ADD', where ⟨n⟩ is the number of products being summed, such as '2ADD' or '4ADD'. Similarly, when an instruction subtracts pairs of products horizontally, the subtractions are denoted in the instruction name by '2SUB'.

Such horizontal reductions, whether by addition or subtraction, inherently reduce the number of lanes by the specified factor ⟨n⟩, causing the element width for the computed result to grow by the

same factor. When the instruction performs ordinary multiplications, denoted in the instruction name by the single letter 'M', then these multiplications are implicitly *widening* ones, generating full-width products. In that case, it is the full-width products that are actually summed or subtracted across lanes.

Some examples of packed-SIMD instructions that multiply and add/subtract the products horizontally are:

PM2ADD.H = P {widening Mul + 2-ADD} .H
PM4ADDASU.B = P {widening Mul + 4-ADD + Accum} SU .B
PM2SUBA.W = P {widening Mul + 2-SUB + Accum} .W
PMQR2ADDA.H = P {MulQR + 2-ADD + Accum} .H
PM2WADD.H = P {widening Mul + 2-WADD} .H

Concerning instruction PMQR2ADDA.H, it was noted earlier that "Q-format" multiplications are by definition not widening but rather always shift down their products as specified in Section 1. Consequently, the subsequent 32-bit horizontal additions and accumulates (2-ADD + Accum, at twice the element width of the source operands) have nearly 16 bits of headroom to avoid overflow.

The last example, PM2WADD.H, involves two widenings of element width, the first implicitly due to the horizontal reduction by a factor of two, and the second explicitly due to the reduction additions themselves (WADD). Because the additions explicitly widen also, the destination operand is an even-odd register pair as explained in Section 3.2.

A different group of packed-SIMD instructions perform summations across all lanes without any multiplications.

Packed-SIMD instructions with full horizontal summation: For packed-SIMD instructions that sum values across all lanes, the horizontal sum may be denoted in the instruction name by 'REDSUM' for *reduction sum*, or, if the horizontal sum is part of a chain of operations, by the shorter abbreviation of just 'SUM'. The use of the name 'REDSUM' or 'SUM' indicates an explicit intention to sum across all lanes, not just by smaller groups of elements as with '2ADD' or '4ADD'. The destination operand is necessarily a scalar (not a SIMD array).

Examples of this last rule are these instructions:

PREDSUM.H = P REDSUM .H
PDIFSUMAU.B = P {DIF + SUM + Accum} U .B

The V extension has several instructions that reduce across all vector elements, with names of the form VRED⟨op⟩.VS for varioius reduction operations ⟨op⟩. One instance of these instructions is VREDSUM.VS for summing all elements of a vector.

Name	Function at odd positions	Function at even positions
PAS.⟨*S*⟩X	addition	subtraction
PSA.⟨*S*⟩X	subtraction	addition
PSAS.⟨*S*⟩X	saturating addition	saturating subtraction
PSSA.⟨*S*⟩X	saturating subtraction	saturating addition
PAAS.⟨*S*⟩X	averaging addition	averaging subtraction
PASA.⟨*S*⟩X	averaging subtraction	averaging addition

Table 2: Packed-SIMD instructions that peform additions in half the element positions and subtractions in the other half. As usual, ⟨*S*⟩ is the element width, 'B', 'H', or 'W'.

3.6 Packed-SIMD instructions with exchanged elements of second operand

The P extension has a few packed-SIMD instructions that swap adjacent elements of the second source operand before then performing a function of a kind already covered by earlier subsections.

Packed-SIMD instructions with exchanged elements of second operand: If a packed-SIMD instruction first swaps pairs of elements of the second source operand before performing the rest of its computation, the instruction name has the usual form for a packed-SIMD instruction but with an 'X' added to the end to visually represent the swapping of elements, like so:

P ⟨*operations*⟩ . ⟨*S*⟩ X

The following are examples of instructions of this type:

PM2ADDA.HX = P {widening Mul + 2-ADD + Accum} .HX
PM2SUB.HX = P {widening Mul + 2-SUB} .HX

Apart from the 'X' at the end, the names of these two examples follow the rules specified in Section 3.5 for instructions with horizontal addition or subtraction.

There is a special category of instructions that swap pairs of elements of the second source operand and then perform additions in half the element positions and subtractions in the other half. All forms of these instructions are listed in Table 2.

Some obvious examples are

PSA.HX = P {Sub (odd positions), Add (even positions)} .HX
PSAS.HX = P {SAdd (odd positions), SSub (even positions)} .HX