# RISC-V Proposed Extension for 32-bit Posits

**John L. Gustafson**

**8 June 2018**

The extension of RISC-V to support 32-bit posits (2-bit *es* exponent value) generally should follow the form of the "F" Standard Extension for Single-Precision Floating-Point. Some features map perfectly; some can be eliminated as unnecessary; a few functions need to be added, such as the quire operations. The following goes through Chapter 6 of the RISC-V Instruction Set Manual and comments on each section.

## 6.1 Register State

Instead of 32 floating-point registers **`f0–f31`**, a posit extension should have 16 posit registers **`p0–16`** and a 512-bit quire register **`q`**. The quire register can also function as posit registers **`p16–p31`**. No "mode bits" or other protections are needed to indicate how the upper half of the register memory is used, as a quire or as 16 posit registers; it is up to the programmer (or the compiler) to manage how those bits are interpreted, and to protect their state (by moving them to stack or using other memory management methods) if an instruction can overwrite data that will be needed subsequently. This provides the necessary functionality without a significant increase in the space needed for the registers.

## 6.2 Posit Control and Status Register

With posits, there is no need for this register. There is only one rounding mode; there are no flags for inexact, overflow, underflow, or posit division by zero. The one error exception, NaR (Not a Real) is silent. The few flags that are needed are the same as those of integer operations, which presumably also flags division by zero. The recommendation is that posit operations affect the same flags as those used for 2's complement signed integers, reducing the logic and storage needed.

An idea to consider, instead, is a pair of 32-bit registers **`maxmag`** and **`minmag`** that can be reset to the 0 at any time, after which they record the largest magnitude and smallest magnitude posit that results from any operation. If the hardware can automatically update these registers without a significant expense of energy or other costs, this can greatly assist a developer or user to know when a calculation has strayed into regions of very low accuracy. If hardware support for registers **`maxmag`** and **`minmag`** is too high, then this function can instead be supported by debugger software.

## 6.3 NaN Generation and Propagation

Posits replace “NaN” with “NaR”; this helps indicate that posits are not the same as floats, and corrects the IEEE 754 mistake of labeling imaginary numbers as not being numbers. They are not *real* numbers.
One important distinction between NaR and the IEEE 754 “NaN” concept is in comparisons. A NaR tests as equal to itself just like any other bit pattern. It maps to the most negative 2’s complement signed integer, and if used it comparison operations, it tests as less than all other posits. In this way, there is no need for posit comparison operations. All the operations of 2’s complement 32-bit integers suffice for comparison. There is no “unordered” concept with posits as there is with NaN.

## 6.4 Single-Precision Load and Store Instructions

Instructions are renamed PLW for loading a posit register from memory and PSW to store a posit register in memory but otherwise behave like single-precision float registers.

However, posit arithmetic needs two additional instructions: QL loads the quire register from memory, and QW writes the quire register to memory. A similar base+offset system can be used for the quire. Quire locations in memory should be integer multiples of 64 bytes (512 bits) for alignment.

## 6.5 Single-Precision Posit Computational Instructions

Posit instructions map to the floating-point instructions except for name: PADD.S, PSUB.S, PMUL.S, PDIV.S, PMIN.S, PMAX.S, and PSQRT.S. Note that PMIN.S and PMAX.S should behave exactly the same as if the arguments were 2’s complement signed integers, so there is no need for extra hardware for those instructions.

The 2-bit encoding *fmt* should change meaning for standard posits, which range from 8-bit to 64-bit precision.

| *Fmt* field | Mnemonic | Meaning |
|---|---|---|
| 00 | B | 8-bit (byte) precision |
| 01 | H | 16-bit (half) precision |
| 10 | S | 32-bit (single) precision |
| 11 | D | 64-bit (double) precision |

There is no need for a rounding mode, so the *rm* field is unused, or perhaps can be repurposed if bits are needed to accommodate quire instructions.

The four fused operations FMADD.S, FMSUB.S, FNMSUB.S, and FNMADD.S are replaced by quire instructions, which only require at most two register fields *rs1* and *rs2* instead of three:

QMADD.S : $q \longleftarrow q + rs1 \times rs2$

QMSUB.S : $q \longleftarrow q - rs1 \times rs2$
QCLR.S : $q \longleftarrow 0$
QNEG.S : $q \longleftarrow -q$

The quire behaves as a fixed-point register where the least-significant 240 bits are the fractional part; sign is indicated using 2's complement. The quire stores NaR as posits do: a **1** in the MSB and **0** for all the other bits. A NaR quire state remains NaR until QCLR.S is called. If *rs1* or *rs2* are NaR posits in a QMADD.S or QMSUB.S operation, the quire becomes NaR in its format. A QNEG.S automatically leaves NaR unchanged just as it leaves 0 unchanged, so no exception is required for it to handle NaR.

The QMADD.S and QMSUB.S perform exact arithmetic with the full product of *rs1* and *rs2* shifted to the position in the quire that, in fixed point, represents the power-of-2 scaling in the mathematical result of the product of two posits. Note that there is no need for a QADD.S or QSUB instruction, since these can be performed simply by using QMADD.S or QMSUB.S with one of the posit arguments set to 1.

We also need a conversion instruction to explicitly round the quire and store in a posit register:

QROUND.S: $rs1 \longleftarrow q$

Rounding is by the usual rules (round to nearest, tie to even, where "nearest" means closest value when rounding fraction bits and closest logarithm when rounding exponent bits).

Note that this system obviates the need for the fused float operations, since a fused multiply-add FMADD.S(*rs1*, *rs2*, *rs3*) is numerically identical to

QCLR.S
QMADD.S(*rs1*, *rs2*)
QMADD.S(*rs3*, 1)
QROUND.S(*rs4*)

but the quire is capable of summing posits or products of posits (dot products) without rounding, up to a very large number of operands, providing much closer adherence to the associative and distributive laws of algebra than floats.

## 6.6 Single-Precision Posit Conversion and Move Instructions

The previous section already described the need to round a quire to a posit, which in a way is conversion instruction; formally, that description might be better placed here.

The posit conversions to integers, and the sign injection instructions, should do exactly what the float instructions do but with posit instead of float operands. The posit for zero is treated as having positive sign, and the posit for NaR is treated as having negative sign. It is the programmer's responsibility to use this convention in a sensible way; no exceptions need to be raised when treating unsigned values 0 and NaR as if they are signed. Hardware logic for negation by 2's complement arithmetic (reverse all bits and add 1 to the LSB, ignoring any carry from the MSB) automatically leaves 0 and NaR unchanged, so no exception logic is needed.

Note that instructions to convert integers to posits are needed, just as they are for floats. If a 64-bit integer has magnitude greater than *maxpos*, it is converted to signed *maxpos*.

## 6.7 Single-Precision Posit Compare Instructions

As previously mentioned, integer comparisons are identical to posit comparisons. Unlike floats, there are no Invalid Operation comparisons when one of the inputs is NaR.

## 6.8 Single-Precision Posit Classify Instructions

This is much less necessary with posits than it is for floats and can be eliminated. There are no subnormals, different kinds of NaN values, infinities, or different kinds of zero, leaving only

*rs1* is 0
*rs1* is NaR
*rs1* is negative
*rs1* is positive

the tests for which are quite trivial and as fast as calling a classify instruction and then examining the mask it produces.