OPF ISA WG External RFC ls005.xlen v1: XLEN </>

- RFC Author: Luke Kenneth Casson Leighton.
- RFC Contributors/Ideas: Jacob Lifshay, Toshaan Bharvani
- Funded by NLnet under the NGI Zero Entrust EU Horizon Europe Grant 101069594
- https://libre-soc.org/openpower/sv/rfc/ls005.xlen/
- https://bugs.libre-soc.org/show_bug.cgi?id=1062
- https://git.libre-soc.org/?p=openpower-isa.git;a=tree;f=openpower/isa;hb=HEAD
- https://git.openpower.foundation/isa/PowerISA/issues/104

Severity: Major

Status: New

Date: 22 Dec 2022 v2 TODO

Target v3.2B

Books and Section affected:

```
Everything (in a consistent, regular and systematic fashion)
```

Summary

```
Exactly as is already done in RISC-V, convert the entire use of 64-bit hard-coding to "XLEN".
Exactly as is in RISC-V, options then include PowerISA-32, PowerISA-64 and PowerISA-128.
Unlike in RISC-V, the concept of PowerISA-16 and PowerISA-8 is also floated, for Embedded,
AI, Edge, Processing-in-Memory, Distributed Computing and other purposes.
```

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Entirely new processors, entirely new markets.
```

Impact on software:

```
Massive but regular, consistent, and systematic.
```

Keywords:

```
XLEN
```

Motivation

The Power ISA is far too massive, making it wholly unsuited for Embedded markets and adversely impacting its reach and potential. The RISC paradigm it is based on has gone too far into PackedSIMD (128-bit). Fixing this is relatively and conceptually straightforward: allow 32-bit and even 16-bit and 8-bit implementations, and use the opportunity to allow future Scalar 128-bit implementations in the exact same strategic way that RISC-V has RV128.

Register files are redefined to XLEN width but are permitted to "group" registers together to create 16-bit, 32-bit and 64-bit addresses. In this way, the limitations of what would otherwise restrict the usefulness of a severely-targetted application-specific processor may be overcome in order to make it still possible to (at reduced performance) still run general-purpose applications. AI application-specific processing or other Processing-In-Memory or other specialist design therefore may for example focus a balance of raw computing power heavily onto 8-bit or 16-bit computation, but still gain the benefit of the Power ISA and everything it brings. Contrast this with the more "normal" approach of creating heavily-focussed specialist "AI" Engines incapable of Turing-completeness and the benefits are clear.

Note 1: SVP64 **requires** this change as a 100% critical dependency. SIMD back-end ALUs process Vectors of "Elements" at 8, 16 and 32-bit (and 64-bit), read from, processed, and returned to, the standard **Scalar** Register Files, with byte-level write-enable lines. The proposal is therefore made as an opportunity for others interested in Scalar ISA 8/16/32-bit (and future 128-bit variants of Scalar Power ISA) to take **and complete** that work in an incremental fashion, without having to be faced with a massive bulk and body of work as a prerequisite.

Examples include that whilst an SVP64 Prefixed "'lbz"' instruction ("'sv.lbz"') is well-defined and has strict well-defined behaviour, a pure **Scalar-only** (non-SVP64) over-ridden"'lbz"' instruction has not been so well-defined, and would require a Stakeholder interested in 8/16/32-bit (and future 128-bit) to think through the implications and incrementally submit further OPF ISA RFCs. With RISC-V **already having done this type of work** it is not technically difficult: it just requires another Stakeholder to do it.

Note 2: one alternative to this proposal, as far as SVP64 is concerned, is to literally duplicate the entirety of Chapters 3 and 4 Book III, and to create - and then maintain - multiple identical copies of the instructions including identical copies of the pseudocode except for substitution of occurrences of "64" with a "32" variant, "16" variant, "8" variant (and future "128" variant), and so on. This would add over 700 additional pages to the Power ISA Specification and it should be clear that it would become a maintenance nightmare.

Another alternative is to poison and irredemably damage the Power ISA (as a powerful and lean RISC ISA) by adding several hundred (close to 1,000) additional specific 8-bit, 16-bit and 32-bit (and in future 128-bit) Scalar instructions. Given that the 32-bit Opcode Allocation Space is already under pressure such a move would be extremely unwise for that reason alone.

Changes

For all pseudocode right across the board in all Scalar operations, replace hard-coded "64" with "XLEN". **This work is already underway as sponsored by NLnet in the Libre-SOC Power ISA Pseudocode**. The default is obviously recommended to be "XLEN=64" in order to create zero disruption.

Definitions of the Register File(s) for GPR and FPR are then changed to be "XLEN" wide. However, for Embedded purposes (XLEN=32/16/8), an SPR controls whether (and how many) sequentially-grouped registers are taken together to create 16-bit, 32-bit and 64-bit addresses (depending on application need). GPR is obvious, FPR is quirky. SVP64 redefines FP ops (those not ending in "s") to be "full width" and all ops ending in "s" to be "half of the full width".

- XLEN=64 keeps FPR "full width" exactly as presently defined, and "half width" exactly as presently defined.
- XLEN=32 overrides FPR "full width" operations to full BFP32, and "half width" to be "BFP16 stored in an BFP32"
- XLEN=16 redefines FPR "full width" operations to full IEEE BFP16 and leaves "half width" RESERVED (there is no IEEE version of FP8).
- XLEN=8 redefines FPR "full width" operations to bfloat16 and leaves "half width" RESERVED.

Examples </>

pseudocode examples demonstrating modification. </>

before for popcntb:

```
do i = 0 to 7
   n <-  0
   do j = 0 to 7
      if (RS)[(i*8)+j] = 1 then
          n <- n+1
   RA[(i*8):(i*8)+7] <-  n
```

after:

```
do i = 0 to ((XLEN/8)-1)
   n <-  0
   do j = 0 to 7
      if (RS)[(i*8)+j] = 1 then
          n <- n+1
   RA[(i*8):(i*8)+7] <-  n
```

Here as the instruction's intent is to count bytes, and RA contains on a per-byte basis a SIMD-style count of each byte's 1s, it becomes possible to simply count less bytes.

Should it be more useful to redefine popcntb in terms of always returning eight results? For example `sv.popcntb/w=16` to return 8 2-bit counts of the number of bits in each 2-bit group in RS?

no modification needed, but function changes </>

For the `addi` instruction there is no apparent change:

```
RT <- (RA|0) + EXTS(SI)
```

However behind the scenes, RA is XLEN bits wide, therefore EXTS performs an increase in bitlength not to exactly 64 but to XLEN. Obviousy for XLEN=16 there is no sign-extension, and for XLEN=8 truncation of `SI` will occur. Illustrates that there are subtle quirks involved, requiring some thought.

The reason for keeping as many bits of the Immediate as possible should be clear.

Compare Ranged Byte (cmprb BF,L,RA,RB) </>

```
    src1    <- EXTZ((RA)[XLEN-8:XLEN-1])
    src21hi <- EXTZ((RB)[XLEN-32:XLEN-23])
    src21lo <- EXTZ((RB)[XLEN-24:XLEN-17])
    src22hi <- EXTZ((RB)[XLEN-16:XLEN-9])
    src22lo <- EXTZ((RB)[XLEN-8:XLEN-1])
    if L=0 then
       in_range <-  (src22lo  <= src1) & (src1 <=  src22hi)
    else
       in_range <- (((src21lo  <= src1) & (src1 <=  src21hi)) |
                    ((src22lo  <= src1) & (src1 <=  src22hi)))
    CR[4*BF+32] <- 0b0
    CR[4*BF+33] <- in_range
    CR[4*BF+34] <- 0b0
    CR[4*BF+35] <- 0b0
```

Compare Ranged Byte takes either one or two ranges from RB as individual bytes, thus requiring a minimum 16-bit (32-bit when L=1) operand RB. src1 on the other hand is only 8-bit long: the first byte of RA.

Therefore a little more thought is required. Should this simply be UNDEFINED behaviour when XLEN=8/16 and L=1? When XLEN=16, L=0 the instruction is still valid. Would it be costly at the Decoder?

Trap Word Immediate </>

Like FP Single operations there also exist operations at "half of regfile width" in the Integer realm. They are discernable with the designation "Word" in their title, such as "Trap WORD Immediate".

```
a <- EXTS((RA)[XLEN/2:XLEN-1])
if (a < EXTS(SI)) & TO[0]  then TRAP
if (a > EXTS(SI)) & TO[1]  then TRAP
if (a = EXTS(SI)) & TO[2]  then TRAP
if (a <u EXTS(SI)) & TO[3] then TRAP
if (a >u EXTS(SI)) & TO[4] then TRAP
```

Here, EXTS receives **half** of the bits of its input register operand, RA. Note this is **not** "32 bit because a Word is 32-bit". The definition "Trap Word Immediate" has to be replaced with "Trap Half-register-width Immediate" but this is very clumsy.

When XLEN=8 "half register width" is clearly 4 bit, thus the LSB nibble is tested, but still sign-extended for comparison against the 16-bit signed immediate.

Extend Sign byte/half/word </>

This instruction can be redefined again in terms of:

- "Word" meaning "Half of register width"
- "Half-word" meaning "Quarter of register width"
- "Byte" meaning "One-eighth of register width"

And a table results as follows:

```
XLEN=8:
            extsb: 1-bit -> 8-bit sign extension
            extsh: 2-bit -> 8-bit sign extension
            extsw: 4-bit -> 8-bit sign extension
XLEN=16:
            extsb: 2-bit -> 16-bit sign extension
            extsh: 4-bit -> 16-bit sign extension
            extsw: 8-bit -> 16-bit sign extension
XLEN=32:
            extsb: 4-bit -> 32-bit sign extension
            extsh: 8-bit -> 32-bit sign extension
            extsw: 16-bit -> 32-bit sign extension
XLEN=64:
            extsb: 8-bit -> 64-bit sign extension
            extsh: 16-bit -> 64-bit sign extension
            extsw: 32-bit -> 64-bit sign extension
```

If the instructions were kept as presently defined then there is a loss of functionality and opportunity:

```
XLEN=8: # completely wasted opportunity
            extsb: 8-bit  -> 8-bit does nothing
            extsh: 16-bit -> 8-bit truncates
            extsw: 32-bit -> 8-bit truncates
XLEN=16: # wasted 2/3 of encoding
            extsb: 8-bit  -> 16-bit sign extension
            extsh: 16-bit -> 16-bit does nothing
            extsw: 32-bit -> 16-bit truncates
XLEN=32: # wasted 1/3 of encoding
            extsb: 8-bit  -> 32-bit sign extension
            extsh: 16-bit -> 32-bit sign extension
            extsw: 32-bit -> 32-bit does nothing
XLEN=64: # unchanged (default) behaviour
            extsb: 8-bit  -> 64-bit sign extension
            extsh: 16-bit -> 64-bit sign extension
            extsw: 32-bit -> 64-bit sign extension
```

The RTL for `extsb` becomes:

```
in <- (RA)[XLEN-8:XLEN-1] # extract first byte
if XLEN = 8  then RT <- in[7] * 8             # 1->8
if XLEN = 16 then RT <- in[6] * 15 || in[7]   # 2->16
if XLEN = 32 then RT <- in[4] * 29 || in[5:7] # 4->32
if XLEN = 64 then RT <- in[0] * 57 || in[1:7] # 8->64
```

And `extsh` and `extsw` follow similar logic. Interestingly there is no loss of functionality compared to keeping `extsb` always as "byte sign-extending" and ironically the loss of opportunity *is* to keep `extsb` the same (extend *byte* regardless of XLEN).

[[!tag opf_rfc]]