Additions to The PowerPC Instruction Set Manual
Libre-SOC Extensions
Document Version 2020-08-28-*draft*

This document is draft

This means that: it is not complete, it may contain outrageous errors, it may knot bee speled write, parts may be in the wrong order, updates will come at strange intervals and may make things worse. No liability will be accepted for any use of the draft document contents.

Editor: Alain D D Williams[1]
[1]Parliament Hill Computers Ltd,
addw@phcomp.co.uk
Friday 28th August, 2020

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Luke Kenneth Casson Leighton, Jacob R Lifshay, Alain D D Williams

Please cite as: "The PowerPC Instruction Set Additions, Document Version 2020-08-28-*draft*", Editor Alain Williams, Libre-SOC, Friday 28th August, 2020.

This document is available at the location below. It will be updated occasionally and might not be the same as the current git sources.

https://ftp.libre-soc.org/power-spec-draft.pdf

Preface

This document describes the Libre-SOC ISAMUX additions to the PowerPC architecture.

Thornton: [20]

Contents

Preface **i**

1 Introduction **1**

1.1 Why has Libre-SOC chosen PowerPC ? 1

1.1.1 Summary 1

1.1.2 One CPU multiple ISAs 2

1.1.3 About Libre-SOC Commercial Project 3

2 Conventions used in this document **5**

3 ISAMUX **7**

3.1 Hypothetical Format 8

3.2 Namespaces are permitted to swap to new state 9

3.3 Privileged Modes / Traps 10

3.4 Alternative RVC 16 Bit Opcode meanings 11

3.5 FAQ 11

3.5.1 Why not have TRAP-ISANS as a vector table, matching mtvec? 11

3.5.2 Is this like MISA ? 12

3.5.3 What happens if this scheme is not adopted? Why is it better than leaving things well alone? 12

3.5.4 Surely it's okay to just tell people to use 48-bit encodings? 13

3.5.5 Why not leave this to individual custom vendors to solve on a case by case basis? 13

3.5.6 Why ISAMUX / ISANS has to be WLRL and mandatory trap on illegal writes 14
3.5.7 Why WARL will not work and why WLRL is required 17
3.5.8 Is it strictly necessary for foreign archs to switch back? 17
3.5.9 Can we have dynamic declaration and runtime declaration of capabilities? . . 17
3.6 Open Questions 18
3.6.1 is the ISANS CSR a 32 or XLEN bit value? 18
3.6.2 Is the ISANS a flat number space or should some bits be reserved for use as flags? 18
3.6.3 Should the ISANS space be partitioned between reserved, custom with registration guaranteed non clashing, custom, very likely non clashing? 18
3.6.4 Should only compiler visible/generated constant setting with CSRRWI and/or using a clearly recognisable LI/LUI be accommodated or should dynamic setting be accommodated as well? 19
3.6.5 How should the ISANS be (re)stored in a trap and in context switch? 19
3.6.6 Should the mechanism accommodate "foreign ISA's" and if so how does one restore the ISA. 19
3.6.7 Where is the default ISA stored and what is responsible for what it is after . 19
3.6.8 If the ISANS is just bits of an instruction that are to be prefixed by the cpu, can those bits contain immediates? Register numbers? 20
3.6.9 How does the system indicate a namespace is not recognised? Does it trap or can/must a recoverable mechanism be provided? 20
3.6.10 What are the security implications? Can some ISA namespaces be set by user space? 20
3.6.11 Does the validity of an ISA namespace depend on privilege level? If so how? 20

4 SimpleV Prefix Proposal – v0.3 23

4.1 Options 23
4.2 Half-Precision Floating Point (FP16) 24
4.3 Compressed Instructions 24
4.4 48-bit Prefixed Instructions 24
4.5 64-bit Prefixed Instructions 24
4.6 48-bit Instruction Encodings 24

4.7 64-bit Instruction Encodings 25
4.8 VLtyp field encoding 26
4.9 vs#/vd Fields' Encoding 28
4.10 Vector Register Number Encoding 28
4.11 Load/Store Kind (lsk) Field Encoding 28
4.12 Sub-Vector Length (svlen) Field Encoding 29
4.13 Predication (pred) Field Encoding 30
4.14 Twin-predication (tpred) Field Encoding 30
4.15 Integer Element Type (itype) Field Encoding 31
4.16 Signedness Decision Procedure 31
4.17 Vector Type and Predication 5-bit (vtp5) Field Encoding 32
4.18 Vector Integer Type and Predication 6-bit (vitp6) Field Encoding 32
4.19 48-bit Instruction Encoding Decision Procedure 32
4.20 CSR Registers 34
4.21 Additional Instructions 34
4.22 Questions 35
4.23 TODO 35

Glossary **39**

Chapter 1

Introduction

1.1 Why has Libre-SOC chosen PowerPC ?

For a hybrid CPU-VPU-GPU, intended for mass-volume adoption in tablets, netbooks, chromebooks and industrial embedded (SBC) systems, our choice was between Nyuzi, MIAOW, RISC-V, PowerPC, MIPS and OpenRISC.

Of all the options, the PowerPC architecture is more complete and far more mature. It also has a deeper adoption by Linux distributions.

Following IBM's release of the Power Architecture instruction set to the Linux Foundation in August 2019 the barrier to using it is no more than that of using RISC-V. We are encouraged that the OpenPOWER Foundation is supportive of what we are doing and helping, e.g by putting us in touch with people who can help us.

1.1.1 Summary

- We propose the standardisation of the way that the PowerPC Instruction Set Architecture (PPC ISA) is extended, enabling many different flavours within a well supported family to co-exist, long-term, without conflict, right across the board.
- This is about more than just our project. Our proposals will facilitate the use of PPC in novel or niche applications without breaking the PPC ISA into incompatible islands.
- PPC will gain a competitive market advantage by removing the need for separate VPU or GPU functions in RTL or ASICs thus enabling lower cost systems. Libre-SOC's project is to extend the PPC to integrate the GPU and VPU functionality directly as part of the PPC ISA (example: Broadcom VideoCore IV being based around extensions to an ARC core).
- Libre-SOC's extensions will be easily adopted, as the standard GNU/Linux distributions will very deliberately run unmodified on our ISA, including full compatibility with illegal instruction trap requirements.

1.1.2 One CPU multiple ISAs

This is a quick overview of the way that we would like to add changes that we are proposing to the PowerPC instruction set (ISA). It is based on a Open Standardisation of the way that existing **mode switches**, already found in the POWER instruction set, are added:

- FPSCR's **NI** bit, setting non-IEEE754 FP mode
- MSR's **LE** bit (and associated HILE bit), setting little-endian mode
- MSR's **SF** bit, setting either 32-bit or 64-bit mode
- PCR's **compatibility** bits 60-62, V2.05 V2.06 V2.07 mode

[It is well-noted that unless each **mode switch** bit is set, any alternative (additional) instructions (and functionality) are completely inaccessible, and will result in **illegal instruction** traps being thrown. This is recognised as being critically important.]

These bits effectively create multiple, incompatible run-time switchable ISAs within one CPU. They are selectable for the needs of the individual program (or OS) being run.

All of these bits are set by an instruction, that, once set, radically changes the entire behaviour and characteristics of subsequent instructions.

With these (and other) long-established precedents already in POWER, there is therefore essentially conceptually nothing new about what we propose: we simply seek that the process by which such **switching** is added is formalised and standardised, such that we (and others, including IBM itself) have a clear, well-defined standards-non-disruptive, atomic and non-intrusive path to extend the POWER ISA for use in markets that it presently cannot enter.

We advocate that some of **mode-setting** (escape-sequencing) bits be binary encoded, some unary encoded, and that some space marked for **offical** use, some **experimental**, some **custom** and some **reserved**. The available space in a suitably-chosen SPR to be formalised, and recommend the OpenPOWER Foundation be given the IANA-like role in atomically allocating mode bits.

The IANA-like atomic role ensures that new PCR mode bits are allocated world-wide unique. In combination with a mandatory illegal instruction exception to be thrown on any system not supporting any given mode, the opportunity exists for all systems to trap and emulate all other systems and thus retain some semblance of interoperability. (Contrast this with either allocating the same mode bit(s) to two (or more) designers, or not making illegal exceptions mandatory: binary interoperability becomes unachievable and the result is irrevocable damage to POWER's reputation.)

We also advocate to consider reserving some bits as a **countdown** where the new mode will be enabled only for a certain number of instructions. This avoids an explicit need to **flip back**, reducing binary code size. Note that it is not a good idea to let the counter cross a branch or other change in PC (and to throw illegal instruction trap if attempted). However traps and exceptions themselves will need to save (and restore) the countdown, just as the rest of the PCR and other modeswitching bits need to be saved.

Instructions that we need to add, which are a normal part of GPUs, include ATAN2, LOG, NORMALISE, YUV2RGB, Khronos Compliance FP mode (different from both IEEE754 and **NI** mode),

and many more. Many of these may turn out to be useful in a wider context: they however need to be fully isolated behind **mode-setting** before being in any way considered for Standards-track formal adoption.

Some mode-setting instructions are privileged, i.e can only be set by the kernel (e.g 32 or 64 bit mode). Most of the escape sequences that we propose will be (have to be) usable without the need for an expensive system call overhead (because some of the instructions needed will be in extremely tight inner loops).

1.1.3 About Libre-SOC Commercial Project

The Libre-SOC Commercial Product is a hybrid CPU-GPU-VPU intended for mass-volume production. There is no separate GPU, because the CPU is the GPU. There is no separate VPU, because the CPU is the GPU. There is not even a separate pipeline: the CPU pipelines are the GPU and VPU pipelines.

Closest equivalents include the ARC core (which has VPU extensions and 3D extensions in the form of Broadcom's VideoCore IV) and the ICubeCorp IC3128. Both are considered **hybrid** CPU-GPU-VPU processors.

Normal Commercial GPUs are entirely separate processors. The development cost and complexity purely in terms of Software Drivers alone is immense. We reject that approach (and as a small team we do not have the resources anyway).

With the project being Libre - not proprietary and secretive and never to be published, ever - it is no good having the extensions as **custom** because **custom** is specifically for the cases where the augmented toolchain is never, under any circumstances, published and made public by the proprietary company (and would never be accepted upstream anyway). For business commercial reasons, Libre-SOC is the total opposite of this proprietary, secretive approach.

Therefore, to meet our business objectives:

- As shown from Nyuzi and Larrabee, although ideally suited to high performance compute tasks, a **traditional** general-purpose full IEEE754-compliant Vector ISA (such as that in POWER9) is not an adequate basis for a commercially competitive GPU. Nyuzi's conclusion is that using such general-purpose Vector ISAs results in reaching only 25(or requiring 4-fold increase in power consumption) to achieve par with current commercial-grade GPUs.
- We are not going the **traditional** (separate custom GPU) route because it is not practical for a new team to design hardware and spend 8+ man-years on massively complex inter-processor driver development as well
- We cannot meet our objectives with a **custom extension** because the financial burden on our team to maintain a total hard fork of not just toolchains, but also entire GNU/Linux Distros, is highly undesirable, and completely impractical (we know for certain that Redhat would strongly object to any efforts to hard-fork Fedora)
- We could invent our own custom GPU instruction set (or use and extend an existing one, to save a man-decade on toolchain development) however even to switch over to that **Dual ISA** GPU instruction set in the next clock cycle still requires a PCR modeswitch bit in order to avoid needing a full Inter-Processor Bus Architecture like on **traditional** GPUs.

- If extending any instruction set, rather than have a Dual ISA (which needs the PCR modeswitch bit to access it) we would rather extend POWER.
- We cannot **go ahead anyway** because to do so would be highly irresponsible and cause massive disruption to the POWER community.

With all impractical options eliminated the only remaining responsible option is to extend the POWER ISA in an atomically-managed (IANA-style) formal fashion, whilst (critically and absolutely essentially) always providing a PCR compatibility mode that is fully POWER compliant, including all illegal instruction traps.

Chapter 2

Conventions used in this document

- Bits are numbered starting from 0 at the LSB, so bit 3 is 1 in the integer 8.
- Bit ranges are inclusive on both ends, so 5:3 means bits 5, 4, and 3.
- Operations work on variable-length vectors of sub-vectors up to VL in length, where each sub-vector has a length svlen, and svlen elements of type etype.
- The actual total number of elements is therefore svlen times VL.
- When the vectors are stored in registers, all elements are packed so that there is no padding in-between elements of the same vector.
- The register file itself is thus best viewed as a byte-level SRAM that is typecast to an array of etypes
- The number of bytes in a sub-vector, svsz, is the product of svlen and the element size in bytes.

Chapter 3

ISAMUX

A fixed number of additional (hidden) bits, conceptually a **namespace**, set by way of a CSR or other out-of-band mechanism, that go directly and non-optionally into the instruction decode phase, extending (in each implementation) the opcode length to 16+N, 32+N, 48+N, where N is a hard fixed quantity on a per-implementor basis.

Where the opcode is normally loaded from the location at the PC, the extra bits, set via a CSR, are mandatorially appended to every instruction: hence why they are described as "hidden" opcode bits, and as a **namespace**.

The parallels with c++ **using namespace** are direct and clear. Alternative conceptual ways to understand this concept include **escape-sequencing**.

TODO: reserve some bits which permit the namespace **escape-sequence** to be relevant for a fixed number of instructions at a time. Caveat: allowing such a countdown to cross branch-points is unwise (illegal instruction?)

An example of a pre-existing **namespace** switch that has been in prevalent use for several decades (SPARC and other architectures): dynamic runtime selectability of little-endian / big-endian **meaning** of instructions by way of a **mode switch** instruction (of some kind).

That **switch** is in effect a 33rd (hidden) bit that is part of the opcode, going directly into the mux / decode phase of instruction decode, and thus qualifies categorically as a **namespace**. This proposal both formalises and generalises that concept.

3.1 Hypothetical Format

Note that this is a hypothetical format, yet to be decided, where particular attention needs to be paid to the fact that there is an **immediate** version of CSRRW (with 5 bits of immediate) that could save a lot of space in binaries.

```
  3                   2                   1
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|-------------------------------|-------|--------------------|-|
|1 custom custom   custom   custom   custom     | foreignarch |1|
|0 reserved reserved reserved reserved reserved | foreignarch |1|
|custom         | reserved    |           official|B| rvcpage |0|
```

RV Mode

- when bit 0 is 0, **RV** mode is selected.
- in RV mode, bits 1 thru 5 provide up to 16 possible alternative meanings (namespaces) for 16 Bit opcodes. **pages** if you will. The top bit indicates custom meanings. When set to 0, the top bit is for official usage.
- Bits 15 thru 23 are reserved.
- Bits 24 thru 31 are for custom usage.
- bit 6 (**B**) is endian-selection: LE/BE

16 bit page examples:

- 0b0000 STANDARD (2019) RVC
- 0b0001 RVCv2
- 0b0010 RV16
- 0b0011 RVCv3
- ...
- 0b1000 custom 16 bit opcode meanings 1
- 0b1001 custom 16 bit opcode meanings 2
- .....

Foreign Arch Mode

- when bit 0 (the LSB) is 1, **Foreign arch** mode is selected.
- Bits 1 thru 7 are a table of foreign arches.
- when the MSB is 1, this is for custom use.
- when the MSB is 0, bits 1 thru 6 are reserved for 64 possible official foreign archs.

Foreign archs could be (examples):

- 0b0000000 x86_32
- 0b0000001 x86_64
- 0b0000010 MIPS32
- 0b0000011 MIPS64
- ....
- 0b0010000 Java Bytecode
- 0b0010001 N.E.Other Bytecode
- ....
- 0b1000000 custom foreign arch 1
- 0b1000001 custom foreign arch 2
- ....

Note that **official** foreign archs have a binary value where the MSB is zero, and custom foreign archs have a binary value where the MSB is 1.

3.2 Namespaces are permitted to swap to new state

In each privilege level, on a change of ISANS (whether through manual setting of ISANS or through trap entry or exit changing the ISANS CSR), an implementation is permitted to completely and arbitrarily switch not only the instruction set, it is permitted to switch to a new bank of CSRs (or a subset of the same), and even to switch to a new PC.

This to occur immediately and atomically at the point at which the change in ISANS occurs.

The most obvious application of this is for Foreign Archs, which may have their own completely separate PC. Thus, foreign assembly code and RISC-V assembly code need not be mixed in the same binary.

Further use-cases may be envisaged however great care needs to be taken to not cause massive complications for JIT emulation, as the RV ISANS is unary encoded (2$\hat{3}$1 permutations).

In addition, the state information of **all** namespaces has to be saved and restored on a context-switch (unless the SP is also switched as part of the state!) which is quite severely burdensome and getting exceptionally complex.

Switching CSR, PC (and potentially SP) and other state on a NS change in the RISCV unary NS therefore needs to be done wisely and responsibly, i.e. minimised!

To be discussed. Context href=https://groups.google.com/a/groups.riscv.org/d/msg/isa-dev/x-uFZDXiOxY/27QDW5KvBQAJ

3.3 Privileged Modes / Traps

An additional WLRL CSR per priv-level named **LAST-ISANS** is required, and another called **TRAP-ISANS** These mirrors the ISANS CSR, and, on a trap, the current ISANS in that privilege level is atomically transferred into LAST-ISANS by the hardware, and ISANS in that trap is set to TRAP-ISANS. Hardware is **only then** permitted to modify the PC to begin execution of the trap.

On exit from the trap, LAST-ISANS is copied into the ISANS CSR, and LAST-ISANS is set to TRAP-ISANS. **Only then** is the hardware permitted to modify the PC to begin execution where the trap left off.

This is identical to how xepc is handled.

Note 1: in the case of Supervisor Mode (context switches in particular), saving and changing of LAST-ISANS (to and from the stack) must be done atomically and under the protection of the SIE bit. Failure to do so could result in corruption of LAST-ISANS when multiple traps occur in the same privilege level.

Note 2: question - should the trap due to illegal (unsupported) values written into LAST-ISANS occur when the **software** writes to LAST-ISANS, or when the **trap** (on exit) writes into LAST-ISANS? this latter seems fraught: a trap, on exit, causing another trap??

Per-privilege-level pseudocode (there exists UISANS, UTRAPISANS, ULASTISANS, MISANS, MTRAPISANS, MLASTISANS and so on):

```
trap_entry()
{
    LAST-ISANS = ISANS // record the old NS
    ISANS = TRAP_ISANS // traps are executed in "trap" NS
}

and trap_exit:

trap_exit():
{
    ISANS = LAST-ISANS
    LAST-ISANS = TRAP_ISANS
}
```

3.4 Alternative RVC 16 Bit Opcode meanings

Here is appropriate to raise an idea how to cover RVC and future variants, including RV16.

Just as with foreign archs, and you quite rightly highlight above, it makes absolutely no sense to try to select both RVCv1, v2, v3 and so on, all simultaneously. An unary bit vector for RVC modes, changing the 16 BIT opcode space meaning, is wasteful and again has us believe that WARL is the **solution**.

The correct thing to do is, again, just like with foreign archs, to treat RVCs as a **binary** namespace selector. Bits 1 thru 3 would give 8 possible completely new alternative meanings, just like how the Zilog Z80 and the 286 and 386 used to do bank switching.

All zeros is clearly reserved for the present RVC. 0b001 for RVCv2. 0b010 for RV16 (look it up) and there should definitely be room reserved here for custom reencodings of the 16 bit opcode space.

3.5 FAQ

3.5.1 Why not have TRAP-ISANS as a vector table, matching mtvec?

Use case to be determined. Rather than be a global per-priv-level value, TRAP-ISANS is a table of length exactly equal to the mtvec/utvec/stvec table, with corresponding entries that specify the assembly-code namespace in which the trap handler routine is written.

Open question: see https://groups.google.com/a/groups.riscv.org/d/msg/isa-dev/IAhyOqEZoWA/BM0G3J2zBgAJ

```
trap_entry(x_cause)
{
    LAST-ISANS = ISANS // record the old NS
    ISANS = TRAP_ISANS_VEC[xcause] // traps are executed in "trap" NS
}

and trap_exit:

trap_exit(x_cause):
{
    ISANS = LAST-ISANS
    LAST-ISANS = TRAP_ISANS_VEC[x_cause]
}
```

3.5.2 Is this like MISA ?

No.

- MISA's space is entirely taken up (and running out).
- There is no allocation (provision) for custom extensions.
- MISA switches on and off entire extensions: ISAMUX/NS may be used to switch multiple opcodes (present and future), to alternate meanings.
- MISA is WARL and is inaccessible from everything but M-Mode (not even readable).

MISA is therefore wholly unsuited to U-Mode usage; ISANS is specifically permitted to be called by userspace to switch (with no stalling) between namespaces, repeatedly and in quick succession.

3.5.3 What happens if this scheme is not adopted? Why is it better than leaving things well alone?

At the first sign of an emergency non-backwards compatible and unavoidable change to the **frozen** RISCV **official** Standards, the entire RISCV community is fragmented and divided into two:

- Those vendors that are hardware compatible with the legacy standard.
- Those that are compatible with the new standard.

These two communities would be mutually exclusively incompatible. If a second emergency occurs, RISC-V becomes even less tenable.

Hardware that wished to be **compatible** with either flavour would require JIT or offline static binary recompilation. No vendor would willingly accept this as a condition of the standards divergence in the first place, locking up decision making to the detriment of RISCV as a whole.

By providing a **safety valve** in the form of a hidden namespace, at least newer hardware has the option to implement both (or more) variations, **and still apply for Certification**.

However to also allow **legacy** hardware to at least be JIT soft compatible, some very strict rules **must** be adhered to, that appear at first sight not to make any sense.

It's complicated in other words!

3.5.4 Surely it's okay to just tell people to use 48-bit encodings?

Short answer: it doesn't help resolve conflicts, and costs hardware and redesigns to do so. Soft-cores in cost-sensitive embedded applications may even not actually be able to fit the required 48 bit instruction decode engine into a (small, ICE40) FPGA. 48-bit instruction decoding is much more complex than straight 32-bit decoding, requiring a queue.

Second answer: conflicts can still occur in the (unregulated, custom) 48-bit space, which **could** be resolved by ISAMUX/ISANS as applied to the **48** bit space in exactly the same way. And the 64-bit space.

3.5.5 Why not leave this to individual custom vendors to solve on a case by case basis?

The suggestion was raised that a custom extension vendor could create their own CSR that selects between conflicting namespaces that resolve the meaning of the exact same opcode. This to be done by all and any vendors, as they see fit, with little to no collaboration or coordination towards standardisation in any form.

The problems with this approach are numerous, when presented to a worldwide context that the UNIX Platform, in particular, has to face (where the embedded platform does not)

First: lack of coordination, in the proliferation of arbitrary solutions, has to primarily be borne by gcc, Binutils, LLVM and other compilers.

Secondly: CSR space is precious. With each vendor likely needing only one or two bits to express the namespace collision avoidance, if they make even a token effort to use worldwide unique CSRs (an effort that would benefit compiler writers), the CSR register space is quickly exhausted.

Thirdly: JIT Emulation of such an unregulated space becomes just as much hell as it is for compiler writers. In addition, if two vendors use conflicting CSR addresses, the only sane way to tell the emulator what to do is to give the emulator a runtime command line argument.

Fourthly: with each vendor coming up with their own way of handling conflicts, not only are the chances of mistakes higher, it is against the very principles of collaboration and cooperation that save vendors money on development and ongoing maintenance. Each custom vendor will have to maintain their own separate hard fork of the toolchain and software, which is well known to result in security vulnerabilities.

By coordinating and managing the allocation of namespace bits (unary or binary) the above issues are solved. CSR space is no longer wasted, compiler and JIT software writers have an easier time, clashes are avoided, and RISCV is stabilised and has a trustable long term future.

3.5.6 Why ISAMUX / ISANS has to be WLRL and mandatory trap on illegal writes

The namespaces, set by bits in the CSR, are functionally directly equivalent to c++ namespaces, even down to the use of braces.

WARL, by allowing implementors to choose the value, prevents and prohibits the critical and necessary raising of an exception that would begin the JIT process in the case of ongoing standards evolution.

Without this opportunity, an implementation has no reliable guaranteed way of knowing when to drop into full JIT mode, which is the only guaranteed way to distinguish any given conflicting opcode. It is as if the c++ standard was given a similar optional opportunity to completely ignore the **using namespace** prefix!

–

Ok so I trust it's now clear why WLRL (thanks Allen) is needed.

When Dan raised the WARL concern initially a situation was masked by the conflict, that if gone unnoticed would jeapordise ISAMUX/ISANS entirely. Actually, two separate errors. So thank you for raising the question.

The situation arises when foreign archs are to be given their own NS bit. MIPS is allocated bit 8, x86 bit 9, whilst LE/BE is given bit 0, RVCv2 bit 1 andso on. All of this potential rather than actual, clearly.

Imagine then that software tries to write and set not just bit 8 and bit 9, it also tries to set bit 0 and 1 as well.

This **IS** on the face of it a legitimate reason to make ISAMUX/ISANS WARL.

However it masks a fundamental flaw that has to be addressed, which brings us back much closer to the original design of 18 months ago, and it's highlighted thus:

x86 and simultaneous RVCv2 modes are total nonsense in the first place!

The solution instead is to have a NS bit (bit0) that SPECIFICALLY determines if the arch is RV or not. If 0, the rest of the ISAMUX/ISANS is very specifically RV **only**, and if 1, the ISAMUX/ISANS is a **binary** table of foreign architectures and foreign architectures only.

Exactly how many bits are used for the foreign arch table, is to be determined. 7 bits, one of which is reserved for custom usage, leaving a whopping 64 possible **official** foreign instruction sets to be hardware-supported/JIT-emulated seems to be sufficiently gratuitous, to me.

One of those could even be Java Bytecode!

Now, it could **hypothetically** be argued that the permutation of setting LE/BE and MIPS for example is desirable. A simple analysis shows this not to be the case: once in the MIPS foreign NS, it is the MIPS hardware implementation that should have its own way of setting and managing its LE/BE mode, because to do otherwise drastically interferes with MIPS binary compatibility.

Thus, it is officially Not Our Problem: only flipping into one foreign arch at a time makes sense, thus this has to be reflected in the ISAMUX/ISANS CSR itself, completely side-stepping the (apparent) need to make the NS CSR WARL (which would not work anyway, as previously mentioned).

So, thank you, again, Dan, for raising this. It would have completely jeapordised ISAMUX/NS if not spotted.

The second issue is: how does any hardware system, whether it support ISANS or not, and whether any future hardware supports some Namespaces and, in a transitive fashion, has to support **more** future namespaces, through JIT emulation, if this is not planned properly in advance?

Let us take the simple case first: a current 2019 RISCV fully compliant RV64GC UNIX capable system (with mandatory traps on all unsupported CSRs).

Fast forward 20 years, there are now 5 ISAMUX/NS unary bits, and 3 foreign arch binary table entries.

Such a system is perfectly possible of software JIT emulating ALL of these options because the write to the (illegal, for that system) ISAMUX/NS CSR generates the trap that is needed for that system ti begin JIT mode.

(This again emphasises exactly why the trap is mandatory).

Now let us take the case of a hypothetical system from say 2021 that implements RVCv2 at the hardware level.

Fast forward 20 years: if the CSR were made WARL, that system would be absolutely screwed. The implementor would be under the false impression that ignoring setting of **illegal** bits was acceptable, making the transition to JIT mode flat-out impossible to detect.

When this is considered transitively, considering all future additions to the NS, and all permutations, it can be logically deduced that there is a need to reserve a **full** set of bits in the ISAMUX/NS CSR **in advance**.

i.e. that **right now**, in the year 2019, the entire ISAMUX/NS CSR cannot be added to piecemeal, the full 32 (or 64) bits **has** to be reserved, and reserved bits set at zero.

Furthermore, if any software attempts to write to those reserved bits, it **must** be treated just as if those bits were distinct and nonexistent CSRs, and a trap raised.

It makes more sense to consider each NS as having its own completely separate CSR, which, if it does not exist, clearly it should be obvious that, as an unsupported CSR, a trap should be raised (and JIT emulation activated).

However given that only the one bit is needed (in RV NS Mode, not Foreign NS Mode), it would be terribly wasteful of the CSRs to do this, despite it being technically correct and much easier to understand why trap raising is so essential (mandatory).

This again should emphasise how to mentally get one's head round this mind-bendingly complex problem space: think of each NS bit as its own totally separate CSR that every implementor is free and clear to implement (or leave to JIT Emulation) as they see fit.

Only then does the mandatory need to trap on write really start to hit home, as does the need to preallocate a full set of reserved zero values in the RV ISAMUX/NS.

Lastly, I **think** it's ok to only reserve say 32 bits, and, in 50 years time if that genuinely is not enough, start the process all over again with a new CSR. ISAMUX2/NS2.

Subdivision of the RV NS (support for RVCv3/4/5/RV16 without wasting precious CSR bits) best left for discussion another time, the above is a heck of a lot to absorb, already.

3.5.7 Why WARL will not work and why WLRL is required

WARL requires a follow-up read of the CSR to ascertain what heuristic the hardware **might** have applied, and if that procedure is followed in this proposal, performance even on hardware would be severely compromised.

In addition when switching to foreign architectures, the switch has to be done atomically and guaranteed to occur.

In the case of JIT emulation, the WARL **detection** code will be in an assembly language that is alien to hardware.

Support for both assembly languages immediately after the CSR write is clearly impossible, this leaves no other option but to have the CSR be WLRL (on all platforms) and for traps to be mandatory (on the UNIX Platform).

3.5.8 Is it strictly necessary for foreign archs to switch back?

No, because LAST-ISANS handles the setting and unsetting of the ISANS CSR in a completely transparent fashion as far as the foreign arch is concerned. Supervisor or Hypervisor traps take care of the context switch in a way that the user mode (or guest) need not be aware of, in any way.

Thus, in e.g. Hypervisor Mode, the foreign guest arch has no knowledge or need to know that the hypervisor is flipping back to RV at the time of a trap.

Note however that this is **not** the same as the foreign arch executing **foreign** traps! Foreign architecture trap and interrupt handling mechanisms are **out of scope** of this document and MUST be handled by the foreign architecture implementation in a completely transparent fashion that in no way interacts or interferes with this proposal.

3.5.9 Can we have dynamic declaration and runtime declaration of capabilities?

Answer: don't know (yet). Quoted from Rogier:

> "A SoC may have several devices that one may want to directly control with custom instructions. If independent vendors use the same opcodes you either have to change the encodings for every different chip (not very nice for software) or you can give the device an ID which is defined in some device tree or something like that and use that."

Dynamic detection wasn't originally planned: static compilation was envisaged to solve the need, with a table of mvendorid-marchid-isamux/isans being maintained inside gcc / binutils / llvm (or separate library?) that, like the Linux kernel ARCH table, requires a world-wide atomic **git commit** to add globally-unique registered entries that map functionality to actual namespaces.

where that goes wrong is if there is ever a pair (or more) of vendors that use the exact same custom feature that maps to different opcodes, a statically-compiled binary has no hope of executing natively on both systems.

at that point: yes, something akin to device-tree would be needed.

3.6 Open Questions

This section from a post by Rogier Bruisse http://hands.com/ lkcl/gmail_re_isadev_isamux.html

3.6.1 is the ISANS CSR a 32 or XLEN bit value?

This is partly answered in another FAQ above: if 32 bits is not enough for a full suite of official, custom-with-atomic-registration and custom-without then a second CSR group (ISANS2) may be added at a future date (10-20 years hence).

32 bits would not inconvenience RV32, and implementors wishing to make significant altnernative modifications to opcodes in the RV32 ISA space could do so without the burden of having to support a split 32/LO 32/HI CSR across two locations.

3.6.2 Is the ISANS a flat number space or should some bits be reserved for use as flags?

See 16-bit RV namespace "page" concept, above. Some bits have to be unary (multiple simultaneous features such as LE/BE in one bit, and augmented Floating-point rounding / clipping in another), whilst others definitely need to be binary (the most obvious one being **paging** in the space currently occupied by RVC).

3.6.3 Should the ISANS space be partitioned between reserved, custom with registration guaranteed non clashing, custom, very likely non clashing?

Yes. Format TBD.

3.6.4 Should only compiler visible/generated constant setting with CSRRWI and/or using a clearly recognisable LI/LUI be accommodated or should dynamic setting be accommodated as well?

This is almost certainly a software design issue, not so much a hardware issue.

3.6.5 How should the ISANS be (re)stored in a trap and in context switch?

See section above on privilege mode: LAST-ISANS has been introduced that mirrors (x)CAUSE and (x)EPC pretty much exactly. Context switches change uepc just before exit from the trap, in order to change the user-mode PC to switch to a new process, and ulast-isans can - must - be treated in exactly the same way. When the context switch sets ulast-isans (and uepc), the hardware flips both ulast-isans into uisans and uepc into pc (atomically): both the new NS and the new PC activate immediately, on return to usermode.

Quite simple.

3.6.6 Should the mechanism accommodate "foreign ISA's" and if so how does one restore the ISA.

See section above on LAST-ISANS. With the introduction of LAST-ISANS, the change is entirely transparent, and handled by the Supervisor (or Hypervisor) trap, in a fashion that the foreign ISA need not even know of the existence of ISANS. At all.

3.6.7 Where is the default ISA stored and what is responsible for what it is after

Options:

- start up
- starting a program
- calling into a dynamically linked library
- taking a trap
- changing privilege levels

These first four are entirely at the discretion of (and the responsibility of) the software. There is precedent for most of these having been implemented, historically, at some point, in relation to LE/BE mode CSRs in other hardware (MIPSEL vs MIPS distros for example).

Traps are responsible for saving LAST-ISANS on the stack, exactly as they are also responsible for saving other context-sensitive information such as the registers and xEPC.

The hardware is responsible for atomically switching out ISANS into the relevant xLAST-ISANS (and back again on exit). See Privileged Traps, above.

3.6.8 If the ISANS is just bits of an instruction that are to be prefixed by the cpu, can those bits contain immediates? Register numbers?

The concept of a CSR containing an immediate makes no sense. The concept of a CSR containing a register number, the contents of which would, presumably, be inserted into the NS, would immediately make that register a permanent and irrevocably reserved register that could not be utilised for any other purpose.

This is what the CSRs are supposed to be for!

It would be better just to have a second CSR - ISANS2 - potentially even ISANS3 in 60+ years time, rather than try to use a GPR for the purposes for which CSRs are intended.

3.6.9 How does the system indicate a namespace is not recognised? Does it trap or can/must a recoverable mechanism be provided?

It doesn't "indicate" that a namespace is not recognised. WLRL fields only hold supported values. If the hardware cannot hold the value, a trap **MUST** be thrown (in the UNIX platform), and at that point it becomes the responsibility of software to deal with it.

3.6.10 What are the security implications? Can some ISA namespaces be set by user space?

Of course they can. It becomes the responsibility of the Supervisor Mode (the kernel) to treat ISANS in a fashion orthogonal to the PC. If the OS is not capable of properly context-switching securely by setting the right PC, it's not going to be capable of properly looking after changes to ISANS.

3.6.11 Does the validity of an ISA namespace depend on privilege level? If so how?

The question does not exactly make sense, and may need a re-reading of the section on how Privilege Modes, above. In RISC-V, privilege modes do not actually change very much state of the system: the absolute minimum changes are made (swapped out) - xEPC, xSTATUS and so on - and the privilege mode is expected to handle the context switching (or other actions) itself.

ISANS - through LAST-ISANS - is absolutely no different. The trap and the kernel (Supervisor or Hypervisor) are provided the **mechanism** by which ISA Namespace **may** be set: it is up to the

software to use that mechanism correctly, just as the software is expected to use the mechanisms provided to correctly implement context-switching by saving and restoring register files, the PC, and other state. The NS effectively becomes just another part of that state.

Chapter 4

SimpleV Prefix Proposal – v0.3

This proposal is designed to be able to operate without SVorig, but not to require the absence of SVorig. See Specification.

Principle: SVprefix embeds (unmodified) RVC and 32-bit scalar opcodes into 32, 48 and 64 bit RV formats, to provide Vectorisation context on a per-instruction basis.

4.1 Options

The following partial / full implementation options are possible:

- SVPrefix augments the main Specification
- SVPrefix operates independently, without the main spec VL (and MVL) CSRs (in any privilege level)
- SVPrefix operates independently, without the main spec SUBVL CSRs (in any priv level)
- SVPrefix has no support for VL (or MVL) overrides in the 64 bit instruction format (VLtyp=0 as the only legal permitted value)
- SVPrefix has no support for svlen overrides in either the 48 or 64 bit instruction format either (svlen=0 as the only legal permitted value).

All permutations of the above options are permitted, and the UNIX platform must raise illegal instruction exceptions on implementations that do not support each option. For example, an implementation that has no support for VLtyp that sees an opcode with a nonzero VLtyp must raise an illegal instruction exception.

Note that SVPrefix (VLtyp and svlen) has its own STATE CSR, SVPSTATE. This allows Prefixed operations to be re-entrant on traps, and to not affect VBLOCK use of VL or SUBVL.

If the main SpecificationCSRs and features are to be supported (VBLOCK), then when VLtyp or svlen are "default" they utilise the main SpecificationVBLOCK VL and/or SUBVL, and, correspondingly, the main VBLOCK STATE CSR will be updated and used to track hardware loops.

If however VLtyp is set to nondefault, then the SVPSTATE src and destoffs fields are used instead to create the hardware loops, and likewise if svlen is set to nondefault, SVPSTATE's svoffs field is used.

4.2 Half-Precision Floating Point (FP16)

If the F extension is supported, SVprefix adds support for FP16 in the base FP instructions by using 10 (H) in the floating-point format field fmt and using 001 (H) in the floating-point load/store width field.

4.3 Compressed Instructions

Compressed instructions are under evaluation by taking the same prefix as used in P48, embedding that and standard RVC opcodes (minus their RVC prefix) into a 32-bit space. This by taking the three remaining Major "custom" opcodes (0-2), one for each of the three RVC Quadrants. see **discussion ???**.

4.4 48-bit Prefixed Instructions

All 48-bit prefixed instructions contain a 32-bit "base" instruction as the last 4 bytes. Since all 32-bit instructions have bits 1:0 set to 11, those bits are reused for additional encoding space in the 48-bit instructions.

4.5 64-bit Prefixed Instructions

The 48 bit format is further extended with the full 128-bit range on all source and destination registers, and the option to set both SVSTATE.VL and SVSTATE.MVL is provided.

4.6 48-bit Instruction Encodings

In the following table, Rsvd (reserved) entries must be zero. RV32 equivalent encodings included for side-by-side comparison (and listed below, separately).

First, bits 17:0:

Encoding	17	16	15	14	13	12	11:7	6	5:0
P48-LD-type	rd[5]	rs1[5]	vitp7[6]	vd	vs1	vitp7[5:0]		Rsvd	011111
P48-ST-type	vitp7[6]	rs1[5]	rs2[5]	vs2	vs1	vitp7[5:0]		Rsvd	011111
P48-R-type	rd[5]	rs1[5]	rs2[5]	vs2	vs1	vitp6		Rsvd	011111
P48-I-type	rd[5]	rs1[5]	vitp7[6]	vd	vs1	vitp7[5:0]		Rsvd	011111
P48-U-type	rd[5]	Rsvd	Rsvd	vd	Rsvd	vitp6		Rsvd	011111
P48-FR-type	rd[5]	rs1[5]	rs2[5]	vs2	vs1	Rsvd	vtp5	Rsvd	011111
P48-FI-type	rd[5]	rs1[5]	vitp7[6]	vd	vs1	vitp7[5:0]		Rsvd	011111
P48-FR4-type	rd[5]	rs1[5]	rs2[5]	vs2	rs3[5]	vs3 [1]	vtp5	Rsvd	011111

[FIXME: The link to [1] is easily confused with the likes of [5]]

[1] Only vs2 and vs3 are included in the P48-FR4-type encoding because there is not enough space for vs1 as well, and because it is more useful to have a scalar argument for each of the multiplication and addition portions of fmadd than to have two scalars on the multiplication portion.

Table showing correspondance between P48–type and RV32–type. These are bits 47:18 (RV32 shifted up by 16 bits):

Encoding	RV32 Encoding
47:32	31:2
P48-LD-type	RV32-I-type
P48-ST-type	RV32-S-Type
P48-R-type	RV32-R-Type
P48-I-type	RV32-I-Type
P48-U-type	RV32-U-Type
P48-FR-type	RV32-FR-Type
P48-FI-type	RV32-I-Type
P48-FR4-type	RV32-FR4-type

Table showing Standard RV32 encodings:

Encoding	31:27	26:25	24:20	19:15	14:12	11:7	6:2	1:0
RV32-R-type	funct7		rs2[4:0]	rs1[4:0]	funct3	rd[4:0]	opcode	0b11
RV32-S-type	imm[11:5]		rs2[4:0]	rs1[4:0]	funct3	imm[4:0]	opcode	0b11
RV32-I-type	imm[11:0]			rs1[4:0]	funct3	rd[4:0]	opcode	0b11
RV32-U-type	imm[31:12]					rd[4:0]	opcode	0b11
RV32-FR4-type	rs3[4:0]	fmt	rs2[4:0]	rs1[4:0]	funct3	rd[4:0]	opcode	0b11
RV32-FR-type	funct5	fmt	rs2[4:0]	rs1[4:0]	rm	rd[4:0]	opcode	0b11

4.7 64-bit Instruction Encodings

Where in the 48 bit format the prefix is "0b0011111" in bits 0 to 6, this is now set to "0b0111111".

63:48	47:18	17:7	6:0
64 bit prefix	RV32[31:3]	P48[17:7]	0b0111111

- The 64 bit prefix format is below

- Bits 18 to 47 contain bits 3 to 31 of a standard RV32 format
- Bits 7 to 17 contain bits 7 through 17 of the P48 format
- Bits 0 to 6 contain the standard RV 64-bit prefix 0b0111111

64 bit prefix format:

Encoding	63	62	61	60	59:48
P64-LD-type	rd[6]	rs1[6]		Rsvd	VLtyp
P64-ST-type		rs1[6]	rs2[6]	Rsvd	VLtyp
P64-R-type	rd[6]	rs1[6]	rs2[6]	vd	VLtyp
P64-I-type	rd[6]	rs1[6]		Rsvd	VLtyp
P64-U-type	rd[6]			Rsvd	VLtyp
P64-FR-type		rs1[6]	rs2[6]	vd	VLtyp
P64-FI-type	rd[6]	rs1[6]	rs2[6]	vd	VLtyp
P64-FR4-type	rd[6]	rs1[6]	rs2[6]	rs3[6]	VLtyp

The extra bit for src and dest registers provides the full range of up to 128 registers, when combined with the extra bit from the 48 bit prefix as well. VLtyp encodes how (whether) to set SVPSTATE.VL and SVPSTATE.MAXVL.

4.8 VLtyp field encoding

NOTE: VL and MVL below are local to SVPrefix and, if non-default, will update the src and dest element offsets in SVPSTATE, not the main SpecificationSTATE. If default (all zeros) then STATE VL and MVL apply to this instruction, and STATE.srcoffs (etc) will be used.

VLtyp[11]	VLtyp[10:6]	VLtyp[5:1]	VLtyp[0]	comment
0	00000	00000	0	no change to VL/MVL
0	VLdest	VLEN	vlt	VL imm/reg mode (vlt)
1	VLdest	MVL+VL-immed	0	MVL+VL immed mode
1	VLdest	MVL-immed	1	MVL immed mode

Note: when VLtyp is all zeros, the main SpecificationVL and MVL apply to this instruction. If called outside of a VBLOCK or if sv.setvl has not set VL, the operation is "scalar".

Just as in the VBLOCK format, when bit 11 of VLtyp is zero:

- if vlt is zero, bits 1 to 5 specify the VLEN as a 5 bit immediate (offset by 1: 0b00000 represents VL=1, 0b00001 represents VL=2 etc.)
- if vlt is 1, bits 1 to 5 specify the scalar (RV standard) register from which VL is set. x0 is not permitted
- VL goes into the scalar register VLdest (if VLdest is not x0)

When bit 11 of VLtype is 1:

- if VLtyp[0] is zero, both SVPSTATE.MAXVL and SVPSTATE.VL are set to (imm+1). The same value goes into the scalar register VLdest (if VLdest is not x0)
- if VLtyp[0] is 1, SVPSTATE.MAXVL is set to (imm+1). SVPSTATE.VL will be truncated to within the new range (if VL was greater than the new MAXVL). The new VL goes into the scalar register VLdest (if VLdest is not x0).

This gives the option to set up SVPSTATE.VL in a "loop mode" (VLtype[11]=0) or in a "one-off" mode (VLtype[11]=1) which sets both MVL and VL to the same immediate value. This may be most useful for one-off Vectorised operations such as LOAD-MULTI / STORE-MULTI, for saving and restoration of large batches of registers in context-switches or function calls.

Note that VLtyp's VL and MVL are not the same as the main SpecificationVL or MVL, and that loops will alter srcoffs and destoffs in SVPSTATE in VLtype nondefault mode, but the srcoffs and destoffs in STATE, if VLtype=0.

Furthermore, the execution order and exception handling must be exactly the same as in the main spec (Program Order must be preserved)

Pseudocode for SVPSTATE.VL:

```
# pseudocode

regs = [0u64; 128];
vl = 0;

// instruction fields:
rd = get_rd_field();
vlmax = get_immed_field();

// handle illegal instruction decoding
if vlmax > XLEN {
    trap()
}

// calculate VL
if rs1 == 0 { // rs1 is x0
    vl = vlmax
} else {
    vl = min(regs[rs1], vlmax)
}

// write rd
if rd != 0 {
    // rd is not x0
    regs[rd] = vl
}
```

4.9 vs#/vd Fields' Encoding

vs#/vd	Mnemonic	Meaning
0	S	the rs#/rd field specifies a scalar (single sub-vector); the rs#/rd field is zero-extended to get the actual 7-bit register number
1	V	the rs#/rd field specifies a vector; the rs#/rd field is decoded using the Vector Register Number Encoding to get the actual 7-bit register number

[**FIXME: Vector Register Number Encoding should be a link**]

If a vs#/vd field is not present, it is as if it was present with a value that is the bitwise-or of all present vs#/vd fields.

- scalar register numbers do NOT increment when allocated in the hardware for-loop. the same scalar register number is handed to every ALU.
- vector register numbers DO increase when allocated in the hardware for-loop. sequentially-increasing register data is handed to sequential ALUs.

4.10 Vector Register Number Encoding

For the 48 bit format, when vs#/vd is 1, the actual 7-bit register number is derived from the corresponding 6-bit rs#/rd field:

Actual 7-bit register number		
Bit 6	Bits 5:1	Bit 0
rs#/rd[0]	rs#/rd[5:1]	0

For the 64 bit format, the 7 bit register is constructed from the 7 bit fields: bits 0 to 4 from the 32 bit RV Standard format, bit 5 from the 48 bit prefix and bit 6 from the 64 bit prefix. Thus in the 64 bit format the full range of up to 128 registers is directly available. This for both when either scalar or vector mode is set.

4.11 Load/Store Kind (lsk) Field Encoding

vd/vs2	vs1	Meaning
0	0	srcbase is scalar, LD/ST is pure scalar.
1	0	srcbase is scalar, LD/ST is unit strided
0	1	srcbase is a vector (gather/scatter aka array of srcbases). VSPLAT and VSELECT
1	1	srcbase is a vector, LD/ST is a full vector LD/ST.

Notes:

- A register strided LD/ST would require 5 registers. srcbase, vd/vs2, predicate 1, predicate 2 and the stride register.

- Complex strides may all be done with a general purpose vector of srcbases.
- Twin predication may be used even when vd/vs1 is a scalar, to give VSPLAT and VSELECT, because the hardware loop ends on the first occurrence of a 1 in the predicate when a predicate is applied to a scalar.
- Full vectorised gather/scatter is enabled when both registers are marked as vectorised, however unlike e.g Intel AVX512, twin predication can be applied.

Open question: RVV overloads the width field of LOAD-FP/STORE-FP using the bit 2 to indicate additional interpretation of the 11 bit immediate. Should this be considered ?

4.12 Sub-Vector Length (svlen) Field Encoding

NOTE: svlen is not the same as the main spec SUBVL. When nondefault (not zero) SVPSTATE context is used for Sub vector loops. However is svlen is zero, STATE and SUBVL is used instead.

Bitwidth, from VL's perspective, is a multiple of the elwidth times svlen. So within each loop of VL there are svlen sub-elements of elwidth in size, just like in a SIMD architecture. When svlen is set to 0b00 (indicating svlen=1) no such SIMD-like behaviour exists and the subvectoring is disabled.

Predicate bits do not apply to the individual sub-vector elements, they apply to the entire subvector group. This saves instructions on setup of the predicate.

svlen Encoding	Value
00	SUBVL
01	2
10	3
11	4

In independent standalone implementations that do not implement the main Specification, the value of SUBVL in the above table (svtyp=0b00) is set to 1, such that svlen is also 1.

Behaviour of operations that set svlen are identical to those of the main spec. See section on VLtyp, above.

4.13 Predication (pred) Field Encoding

pred	Mnemonic	Predicate Register	Meaning
000	None	None	The instruction is unpredicated
001	Reserved	Reserved	
010	!x9	x9 (s1)	execute vector op[0..i] on x9[i] == 0
011	x9		execute vector op[0..i] on x9[i] == 1
100	!x10	x10 (a0)	execute vector op[0..i] on x10[i] == 0
101	x10		execute vector op[0..i] on x10[i] == 1
110	!x11	x11 (a1)	execute vector op[0..i] on x11[i] == 0
111	x11		execute vector op[0..i] on x11[i] == 1

4.14 Twin-predication (tpred) Field Encoding

Twin-predication (ability to associate two predicate registers with an instruction) applies to MV, FCLASS, LD and ST. The same format also applies to integer-branch-compare operations although it is not to be considered "twin" predication. In the case of integer-branch-compare operations, the second register (if enabled) stores the results of the element comparisons. See Appendix for details.

[FIXME: Appendix above is link to http://libre-riscv.org/simple_v_extension/appendix/]

pred	Mnemonic	Predicate Register	Meaning
000	None	None	The instruction is unpredicated
001	x9,off	src=x9, dest=none	src[0..i] uses x9[i], dest unpredicated
010	off,x10	src=none, dest=x10	dest[0..i] uses x10[i], src unpredicated
011	x9,10	src=x9, dest=x10	src[0..i] uses x9[i], dest[0..i] uses x10[i]
100	None	RESERVED	Instruction is unpredicated (TBD)
101	!x9,off	src=!x9, dest=none	
110	off,!x10	src=none, dest=!x10	
111	!x9,!x10	src=!x9, dest=!x10	

[FIXME: In table above some in col 3 might be vertically joined]

4.15 Integer Element Type (itype) Field Encoding

Signedness [2]	itype	Element Type	Mnemonic in Integer Instructions	Mnemonic in FP Instructions (such as fmv.x)	Meaning (INT may be un/signed, FP just re-sized
Unsigned	01	u8	BU	BU	Unsigned 8-bit
	10	u16	HU	HU	Unsigned 16-bit
	11	u32	WU	WU	Unsigned 32-bit
	00	uXLEN	WU/DU/QU	WU/LU/TU	Unsigned XLEN-bit
Signed	01	i8	BS	BS	Signed 8-bit
	10	i16	HS	HS	Signed 16-bit
	11	i32	W	W	Signed 32-bit
	00	iXLEN	W/D/Q	W/L/T	Signed XLEN-bit

[2] (1, 2) Signedness is defined in Signedness Decision Procedure

Note: vector mode is effectively a type-cast of the register file as if it was a sequential array being typecast to typedef itype[] (c syntax). The starting point of the "typecast" is the vector register rs#/rd.

Example: if itype=0b10 (u16), and rd is set to "vector", and VL is set to 4, the 64-bit register at rd is subdivided into FOUR 16-bit destination elements. It is NOT four separate 64-bit destination registers (rd+0, rd+1, rd+2, rd+3) that are sign-extended from the source width size out to 64-bit, because that is itype=0b00 (uXLEN).

Note also: changing elwidth creates packed elements that, depending on VL, may create vectors that do not fit perfectly onto XLEN sized registry file bit-boundaries. This does NOT result in the destruction of the MSBs of the last register written to at the end of a VL loop. More details on how to handle this are described in the main Specification.

4.16 Signedness Decision Procedure

1. If the opcode field is either OP or OP-IMM, then

   Signedness is Unsigned.

2. If the opcode field is either OP-32 or OP-IMM-32, then

   Signedness is Signed.

3. If Signedness is encoded in a field of the base instruction, [3] then

   Signedness uses the encoded value.

4. Otherwise,

   Signedness is Unsigned.

[3] Like in fcvt.d.l[u], but unlike in fmv.x.w, since there is no fmv.x.wu

4.17 Vector Type and Predication 5-bit (vtp5) Field Encoding

In the following table, X denotes a wildcard that is 0 or 1 and can be a different value for every occurrence.

vtp5	pred	svlen
1XXXX	vtp5[4:2]	vtp5[1:0]
01XXX		
000XX		
001XX	Reserved	

4.18 Vector Integer Type and Predication 6-bit (vitp6) Field Encoding

In the following table, X denotes a wildcard that is 0 or 1 and can be a different value for every occurrence.

vitp6	itype	pred[2]	pred[0:1]	svlen
XX1XXX	vitp6[5:4]	0	vitp6[3:2]	vitp6[1:0]
XX00XX				
XX01XX	Reserved			

[FIXME: spanning cols/rows above]

vitp7 field: only tpred

vitp7	itype	tpred[2]	tpred[0:1]	svlen
XXXXXXX	vitp7[5:4]	vitp7[6]	vitp7[3:2]	vitp7[1:0]

4.19 48-bit Instruction Encoding Decision Procedure

In the following decision procedure, *Reserved* means that there is not yet a defined 48-bit instruction encoding for the base instruction.

1. If the base instruction is a load instruction, then
   (a) If the base instruction is an I-type instruction, then
      i. The encoding is P48-LD-type.
   (b) Otherwise
      i. The encoding is *Reserved*.

2. If the base instruction is a store instruction, then
   (a) If the base instruction is an S-type instruction, then
      i. The encoding is P48-ST-type.
   (b) Otherwise
      i. The encoding is *Reserved*.
3. If the base instruction is a SYSTEM instruction, then
   (a) The encoding is *Reserved*.
4. If the base instruction is an integer instruction, then
   (a) If the base instruction is an R-type instruction, then
      i. The encoding is P48-R-type.
   (b) If the base instruction is an I-type instruction, then
      i. The encoding is P48-I-type.
   (c) If the base instruction is an S-type instruction, then
      i. The encoding is *Reserved*.
   (d) If the base instruction is an B-type instruction, then
      i. The encoding is *Reserved*.
   (e) If the base instruction is an U-type instruction, then
      i. The encoding is P48-U-type.
   (f) If the base instruction is an J-type instruction, then
      i. The encoding is *Reserved*.
   (g) Otherwise
      i. The encoding is *Reserved*.
5. If the base instruction is a floating-point instruction, then
   (a) If the base instruction is an R-type instruction, then
      i. The encoding is P48-FR-type.
   (b) If the base instruction is an I-type instruction, then
      i. The encoding is P48-FI-type.
   (c) If the base instruction is an S-type instruction, then
      i. The encoding is *Reserved*.
   (d) If the base instruction is an B-type instruction, then
      i. The encoding is *Reserved*.
   (e) If the base instruction is an U-type instruction, then
      i. The encoding is *Reserved*.
   (f) If the base instruction is an J-type instruction, then

i. The encoding is *Reserved*.

(g) If the base instruction is an R4-type instruction, then

i. The encoding is P48-FR4-type.

(h) Otherwise

i. The encoding is *Reserved*.

6. Otherwise The encoding is *Reserved*.

4.20 CSR Registers

CSRs are the same as in the main Specification, if associated functionality is implemented. They have the exact same meaning as in the main Specification.

- VL
- MVL
- SVPSTATE
- SUBVL

Associated SET and GET on the CSRs is exactly as in the main spec as well (including CSRRWI and CSRRW differences).

Note that if both VLtyp and svlen are not implemented, SVPSTATE is not required. Also if VL and SUBVL are not implemented, STATE from the main Specificationis not required either.

However if partial functionality is implemented, the unimplemented bits in STATE and SVPSTATE must be zero, and, in the UNIX Platform, an illegal exception MUST be raised if unsupported bits are written to.

SVPSTATE fields are exactly the same layout as STATE:

(31..28)	(27..26)	(25..24)	(23..18)	(17..12)	(11..6)	(5...0)
rsvd	dsvoffs	subvl	destoffs	srcoffs	vl	maxvl

However note that where STATE stores the scalar register number to be used as VL, SVPSTATE.VL actually contains the actual VL value, in an identical fashion to RVV.

4.21 Additional Instructions

- Add instructions to convert between integer types.
- Add instructions to swizzle elements in sub-vectors. Note that the sub-vector lengths of the source and destination won't necessarily match.
- Add instructions to transpose (2-4)x(2-4) element matrices.

- Add instructions to insert or extract a sub-vector from a vector, with the index allowed to be both immediate and from a register (immediate can be covered by twin-predication, register might be, by virtue of predicates being registers)
- Add a register gather instruction (aka MV.X: regfile[rd] = regfile[regfile[rs1]])

subelement swizzle example:

velswizzle x32, x64, SRCSUBVL=3, DESTSUBVL=4, ELTYPE=u8, elements=[0, 0, 2, 1]

4.22 Questions

Moved to the discussion page (link at top of this page)

4.23 TODO

Work out a way to do sub-element swizzling.

[]

ulp=ulp

cite=[13]

book=[18]

Bibliography

[1] Bkm algorithm. Technical report, https://en.wikipedia.org/wiki/BKM_algorithm. The BKM algorithm is a shift-and-add algorithm for computing elementary functions, first published in 1994 by Jean-Claude Bajard, Sylvanus Kla, and Jean-Michel Muller.

[2] *OpenFSI Specification: Field Replaceable Unit Service Interface.* http://openpowerfoundation.org/wp-content/uploads/resources/OpenFSI-spec-100/OpenFSI-spec-20161212.pdf, December 2016. Field replaceable unit Support Interface(FSI) suited to service all chips in acomputer system via a common serial interface.

[3] The design of a cordic-based sine and cosine computer. Technical report, http://www.myhdl.org/docs/examples/sinecomp/, November 2017.

[4] Using a cordic to calculate sines and cosines in an fpga. Technical report, Gisselquist Technology, https://zipcpu.com/dsp/2017/08/30/cordic.html, August 2017.

[5] *OpenFSI Compliance Specification: Field Replaceable Unit Service Interface.* http://openpowerfoundation.org/wp-content/uploads/resources/openpower-fsi-thts-1.0/openpower-fsi-thts-20180130.pdf, January 2018. The purpose of the OpenPOWER FSI Compliance Test Harness and Test Suite (TH/TS) Specificationis to provide the test suite requirements to be able to demonstrate OpenPOWER FSI compliance.

[6] Ray Andraka. A survey of cordic algorithms for fpga based computers. Technical report, Andraka Consulting Group, Inc, http://www.andraka.com/files/crdcsrvy.pdf, 1998. Attempt to survey commonly used functions that may be accomplished using a CORDIC architecture, explain how the algorithms work, and explore implementation specific to FPGAs.

[7] Raymond Chen. The powerpc 600 series, part 1: Introduction. Technical report, Microsoft, https://devblogs.microsoft.com/oldnewthing/20180806-00/?p=99425, August 2018. Windows NT use of PowerPC.

[8] Raymond Chen. The powerpc 600 series, part 2: Condition registers and the integer exception register. Technical report, Microsoft, https://devblogs.microsoft.com/oldnewthing/20180807-00/?p=99435, August 2018. Condition registers and the integer exception register.

[9] Raymond Chen. The powerpc 600 series, part 3: Arithmetic. Technical report, Microsoft, https://devblogs.microsoft.com/oldnewthing/20180808-00/?p=99445, August 2018. Windows NT use of PowerPC.

[10] IEEE Microprocessor Standards Committe. 754-2019 - ieee standard for floating-point arithmetic. Technical report, https://standards.ieee.org/standard/754-2019.html, July

2019. This standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments.

[11] Bruce Dawson. Intel underestimates error bounds by 1.3 quintillion. Technical report, https://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/, October 2014.

[12] Scott Duplichan. Intel overstates fpu accuracy. Technical report, http://notabs.org/fpuaccuracy/, June 2016. FPU inaccuracy is greater than the ulps claimed.

[13] David Goldberg. What every computer scientist should know about floating-point arithmetic. Technical report, https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html, March 1991. This paper presents a tutorial on those aspects of floating-point that have a direct impact on designers of computer systems.

[14] Thomas F. Hain and David B. Mercer. Fast floating point square root. Technical report, https://pdfs.semanticscholar.org/5060/4e9aff0e37089c4ab9a376c3f35761ffe28b.pdf, 2005.

[15] IBM, https://openpowerfoundation.org/?resource_lib=ibm-power-isa-version-2-07-b. *Power ISA Version 2.07B*, June 2016. Specification that describes the architecture used for the IBM POWER8, IBM POWER8 with NVIDIA NVLink™ Technology, the Suzhou Powercore Technology CP1 processor and prior IBM Power Architecture processors.

[16] IBM, https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0. *Power ISA Version 3.0B*, March 2017. Specification that describes the architecture used for the IBM POWER9 processor.

[17] Karen Miller. Floating point representation. Technical report, http://pages.cs.wisc.edu/~markhill/cs354/Fall2008/notes/flpt.apprec.html, 2006. Introduction to floating point.

[18] Juan Segarra, Clemente Rodríguez, Rubén Gran, Luis C. Aparicio, and Víctor Viñals. *ACDC: Small, Predictable and High-Performance Data Cache*. https://dl.acm.org/doi/10.1145/2677093, 2015.

[19] Suraj Jitindar Singh and David Gibson. Implement isa v3.00 radix page fault handler. Technical report, https://github.com/qemu/qemu/commit/d5fee0bbe68d5e61e2d2beb5ff6de0b9c1cfd182, May 2017.

[20] J. E. Thornton. *Design of a computer. The Control Data 6600.* Scott, Forseman and Company, https://archive.computerhistory.org/resources/text/CDC/cdc.6600.thornton.design_of_a_computer_the_control_data_6600.1970.102630394.pdf, 1970.

[21] Sorbonne Université. Free silicon conference. In *Second Free Silicon Conference*, https://wiki.f-si.org/index.php/FSiC2019, 2019.

[22] Baozhou Zhu, Yuanwu Lei, Yuanxi Peng, and Tingting He. Low latency and low error floating-point sine/cosine function based tcordic algorithm. Technical report, https://ieeexplore.ieee.org/document/7784797, December 2016. TCORDIC algorithm, which combines low latency CORDIC and Taylor algorithm, is presented.

Glossary

BE Big Endian. When 2/4/8 bytes are loaded into a 16/32/64 bit register the bytes at **lower** memory addresses are put into **higher – more significant** places in the register. IBM z is BE. See: LE and endian . 8, 39, 40

Binutils GNU Binary Utilities is part of the toolchain used for creating and managing binary objects (compiled code). Used with gcc. See: GNU web site . 13

CPU Central Processing Unit. The brain of a conventional computer that executes general purpose programs. Contrast with GPU and VPU. . 3, 39, 41

CSR Control and Status Register. A special register that records CPU status and processing options. One important option to this project is that the special instructions that we have created will be recognised. . 7, 9, 23, 40

endian Describes in a multi-byte word, which byte contains the most significant bits. Two choices Little Endian LE and and Big Endian BE predominate, but it can be more complicated when a word is made of 4 or more bytes. PowerPC, ARM & SPARC can be either LE or BE. See: Wikipedia . 8, 39, 40

FPGA Field-programmable gate array. An integrated circuit where the circuitry can be reconfigured. See: Wikipedia . 13

FPU Floating Point Unit. The part of the computer that does calculations of data in, probably, IEEE754 format. See: Wikipedia . 40

gcc GNU Compiler Collection. A popular open source compiler for C (& related), Fortran, Ada & Go and able to generate object code for many ISAs including PowerPC. See: Wikipedia . 13, 39

GNU The GNU project is a large collection of free software. It provides many of the core programs that are used by many Linux distributions. See: GNU website . 39

GPU Graphics processing unit. Special purpose processor optimised for graphics and image generation, often able to run in parallel – the same instructions on different data at the same time. Contrast with CPU and VPU. See: Wikipedia . 3, 39

ICubeCorp IC3128 A SoC from ICube that has both CPU and GPU on a single chip. See: CNX Software . 3

IEEE754 A popular standard way of representing and manipulating floating point numbers. Initiated by the Institute of Electrical and Electronics Engineers in 1985. Different precisions from 16 to 256 bits are described. See: FPU Wikipedia . 2, 39

ISA Instruction Set Architecture. An abstract model of a computer a definition that includes: registers, memory access, input/output, data types, CPU instruction set. Everything that is needed to be able to create programs to run on the machine. See: Wikipedia . 39–41

ISAMUX ISA MUX – having the same bits in the ISA mean different things. . 12

JIT Just In Time compilation. Translate when the program runs, only when needed. See: Wikipedia . 12

LE Little Endian. When 2/4/8 bytes are loaded into a 16/32/64 bit register the bytes at **lower** memory addresses are put into **lower − less significant** places in the register. Intel/AMD are LE. See: BE and endian . 8, 39

Linux A free kernel on which free operating systems and specialised environments are built. Linux is found all the way for small, embedded systems, to desktops, to servers and the world's biggest super computers. It runs on many ISAs and supports a huge variety of peripheral devices. Linux was inspired by Unix and is upwards compatible with POSIX. See: Linux Foundation . 39

LLVM A compiler and related toolchain. An open source and able to compileAda, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift able to generate object code for many ISAs including PowerPC. See: Wikipedia . 13

LSB Least Significant Bit. In an integer represented in binary, the bit that has the smallest value. In this document the LSB is called 'bit 0', if this is the only bit set the integer will have the value 1. See: MSB . 8, 40

MISA Multiple Instruction Sets Architecture. The ability to run more than one ISA on the same hardware. A setting in a CSR controls which instructions will be recognised at any time. . 12

MSB Most Significant Bit. In an integer represented in binary, the bit that has the greatest value. If the integer is signed, this bit will make the integer negative if it is 1. In this document the MSB is given the highest number in the integer, eg: in 8 bits it is called 'bit 7'; in 32 bits it is called 'bit 31'. See: LSB . 8, 40

MUX Multiplex, a way of compressing several things into the same data. . 40

PC Program Counter. A register that holds the address of the instruction being executed. . 9, 10

PowerPC A RISC ISA created in 1991 by Apple, IBM and Motorola. The name is a backronym: Performance Optimization With Enhanced RISC Performance Computing, sometimes abbreviated as PPC or called POWER). See: Wikipedia . 1, 39, 40

RISC Reduced Instruction Set Computer. A computer design philosophy that features simple but fast instructions, often with many registers. See: Wikipedia . 40, 41

RISC-V An open sourced RISC ISA started in 2010 at the University of California, Berkeley. See: Wikipedia . 9, 12

SoC System on a Chip. An integrated circuit that has (almost all) the components needed to make a fully running system. See: Wikipedia . 17, 39, 41

SP Stack Pointer. A register that holds the address of the current function stack frame – used for variables local to a function. . 9

SPARC A RISC ISA created by Sun Microsystems. See: Wikipedia . 7, 39

Supervisor Mode A privileged CPU state where the program can execute instructions or otherwise do things that a non-privileged program would not be allowed to do. . 10

ulp Unit in the Last Place. A measure of accuracy of floating point operations. See: Wikipedia and FPbasics [13] . 35, 38

VideoCore IV Low power SoC from Broadcom. ARM CPU that is used in the Raspberry Pi. See: Wikipedia . 3

VPU Video Processing Unit. Similar to a CPU but has extra hardware instructions to speed up things . 3, 39

Zilog Z80 An 8 bit processor produced by the Zilog Inc in 1986. It is compatible with the Intel 8080 processor. See: Wikiedia . 11

Acknowledgement and thanks for some of the glossary text having been taken from: Wikipedia, and other places.