Let’s make a standard

Where are we and how do we move forwards?
From CHERI v8 to CHERI-RISC-V

- CHERI v8 imported into CHERI v9 repo in April 2022

- CHERI v9 changes include
  - CHERI-MIPS removed
  - CHERI-RISC-V now the base architecture
  - CHERI-x86 sketch added

- CHERI-RISC-V changes in CHERI v9 include
  - Merged register-file only, split option removed
  - Mode dependant JALR – JALR.PCC, JALR.MODE
    - Although subsequently removed thanks for fast mode switching
  - Removal of DDC, PCC offsetting
  - Add CGetHigh, CSetHigh for capability creation and querying
  - Add per-privilege enables into menvfg, servcfg CSRs
  - Moving to tag clearing to reduce exception sources
    - This is not a complete list.....
From CHERI-RISC-V v9 to the Codasip Demo

• At Codasip we worked independently to fill in gaps in the specification to allow a product to be built
  • There was no CHERI-RISC-V debug specification (Sdtrig/Sdext)
  • Not all mnemonics were clearly specified
    • E.g. did c.j map to c.cj in capability mode? The semantics don’t change....
    • Missing encodings for 16-bit instructions....
  • Merging the exception priorities with the standard RISC-V ones
  • And various other changes (this is not an exhaustive list)

• We demonstrated the result of this development at the RISC-V summit in Santa Clara, November 2023
The Demo at the RISC-V Summit Nov ‘24

- Carl Shaw and Troy Jones showing the A730-CHERI prototype detecting a buffer overrun on the stack
At Codasip we started working with Cambridge University on the CHERI-RISC-V specification after discussion at the RISC-V Summit in Barcelona in June 2023
  • Although we were thwarted by emails going to spam for quite some time in both directions!

We were already working in the background on a different version of the CHERI specification document
  • Extracting well defined features from CHERI v9
    • Postponing experimental and less well-defined features
    • Defining a stable base architecture
  • Written as an implementation spec
  • Covers all the necessary questions asked by the implementation and verification teams to allow the product to be built
CHERI-RISC-V v0.7.0

• After review with Cambridge, the Codasip CHERI spec document became v0.7.0 on Github
  • See https://github.com/riscv/riscv-cheri/releases/

• The TG was formed in January 2024
  • Chair/vice-chair *still* not confirmed

• And then the real spec work started refining the architecture

• A few examples of the problems we needed to solve are on the next few slides
RV32: Old Capability Format

- The RV32 format poses challenges due to limited encoding space
  - CHERI v9 has
    - 12-bit permissions
    - 8-bit mantissa (encoded as 8 for Base, 6 for Top) – 14-bits
    - 1-bit flag (Mode)
    - 4-bit Otype
    - 1-bit Internal Exponent flag

- A few observations
  - That’s a lot of permission bits
  - Otype has been deferred
  - The mantissa is too short for accurate bounds
  - Mode doesn’t need a whole bit, it’s only relevant for executable caps
  - We want space for future expansion
  - No software defined permissions

- So, let’s rework the format
RV32: Old Permission Bits (12 to 5)

- Divide them into categories, and only allocate 5 in the base extension
  - access_system_regs – kept as ASR
  - permit_store – kept as W
  - permit_load – kept as R
  - permit_execute – kept as X
  - permit_store_cap – combined into C permission (also requires W)
  - permit_load_cap – combined into C permission (also requires R)
    - 5-bits allocated
  - permit_store_local_cap – separate local/global extension
  - global – separate local/global extension
    - Will be allocated later
  - permit_set_CID – deferred
  - permit_unseal – deferred
  - permit_seal – deferred
  - permit_cinvoke – deferred
    - 4-bits saved
RV32: New Format

- Including Mode we want 6 permission bits, but in limited combinations, so we encode this down to 5-bits
- We have a single root capability, but define different quadrants with only 1 valid for PCC
- Mode is ¼ bit overall
- Lots of expansion room – 19/32 are unallocated

<table>
<thead>
<tr>
<th>Encoding[2:0]</th>
<th>R</th>
<th>W</th>
<th>C</th>
<th>X</th>
<th>ASR</th>
<th>Mode</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 0: Non-capability data read/write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>No permissions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td></td>
<td>N/A</td>
<td>Data RO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>Data WO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>Data RW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-7</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reserved bits for future extensions are 0 so new permissions are not implicitly granted.

Quadrant 1: Executable capabilities

<table>
<thead>
<tr>
<th>Encoding[2:0]</th>
<th>R</th>
<th>W</th>
<th>C</th>
<th>X</th>
<th>ASR</th>
<th>Mode</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Mode</td>
<td>Execute + ASR (see Infinite)</td>
</tr>
<tr>
<td>2-3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>Mode</td>
<td>Execute + Data &amp; Cap RO</td>
</tr>
<tr>
<td>4-5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Mode</td>
<td>Execute + Data &amp; Cap RW</td>
</tr>
<tr>
<td>6-7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>Mode</td>
<td>Execute + Data RW</td>
</tr>
</tbody>
</table>

Quadrant 2: Reserved

Reserved bits for future extensions must be 1 so they are implicitly granted.

<table>
<thead>
<tr>
<th>Encoding[2:0]</th>
<th>R</th>
<th>W</th>
<th>C</th>
<th>X</th>
<th>ASR</th>
<th>Mode</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>

Quadrant 3: Capability data read/write


Reserved bits for future extensions must be 1 so they are implicitly granted.

<table>
<thead>
<tr>
<th>Encoding[2:0]</th>
<th>R</th>
<th>W</th>
<th>C</th>
<th>X</th>
<th>ASR</th>
<th>Mode</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>Data &amp; Cap RO</td>
</tr>
<tr>
<td>4-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>N/A</td>
<td>Data &amp; Cap RW</td>
</tr>
</tbody>
</table>
RV32: New Format

- The final encoding has
  - 2 software defined permissions
  - 5 architectural permissions (6 including the Mode bit)
    - With space for more to be added
  - 4 reserved bits (for local/global?)
  - 1 sealed bit
  - T8 gives an extra mantissa bit when the exp is zero, or a 5-bit exp field

![Diagram of capability encoding when MXLEN=32]

*Figure 1. Capability encoding when MXLEN=32*
RV32: Fixing AUIPC+JALR

• AUIPC+JALR sequences could in theory be problematic
  • Due to the short mantissa, and the calculation method for the representable range: in theory it was possible for the intermediate result to be unrepresentable, even though the final result from the JALR is representable
  • This would prevent linker relaxation from working
  • I don’t think this problem has been seen “in the wild”

• The solution?
  • Increase the mantissa length to 10-bits
  • Change the representable range definition so that it is much more closely centred on the bounds. Instead of only guaranteeing 1/8 above or below (due to a 3-bit subtraction) it now does a full 10-bit subtraction
  • Thanks to Peter Rugg for proving the result
  • We also do the same for RV64, just to be consistent
Illegal address handling: background

• This one was particularly tough to solve, but we’re happy with the solution.

• For VM cores implementing Sv48 or Sv39 it is highly desirable to trim the PC and other address registers to save considerable area.
  • Less so for Sv57

• If the upper bits (i.e. [63:39] for Sv39 don’t match) then the address is invalid, and so we can store an invalid flag and only 39-bits of address.

• When expanding the address there’s no guarantee that it will be the same as the original invalid address, RISC-V only requires that they are both invalid, and so will both cause an access fault.
Illegal address handling: the rub!

• Here’s the problem
  • The address changes, which for a CHERI machine is very very bad
    • The new address might be unrepresentable, meaning that replacing the address without specific checks might cause a tagged capability with unexpected properties
  • It’s possible that the original address was in bounds, and the new address is out of bounds
  • The reverse is also possible, the address may change from out of bounds to in bounds
  • The address must have been representable before, but may not be after

• What we don’t want to do is spend expensive hardware on a case which is guaranteed to cause an exception
  • No more bounds check, representable range check logic please

• Note that Legacy Mode must cause an access fault for compatibility
Illegal address handling: the solution

• We need to distinguish between a CHERI core running non-CHERI software, which must cause an **access fault**, and running CHERI software where we can take a **CHERI fault**
  • If the capability metadata is infinity (no CHERI configuration has been done) then simply update the address and this will trigger an access fault
    • The new address is guaranteed to be representable and in-bounds
  • If the capability metadata is *not* infinity then take a CHERI invalid address fault, and do not proceed further
    • *This covers branch targets, jump targets, load/store addresses*

• This has a very useful property – it means that bounds which are in the invalid address space now have no use – so do not need to be compared against.
  • *This means that we need 39-bit comparators for Sv39 or 48-bit for Sv48 instead of 64-bit.*
  • **CHERI just got a bit cheaper in area and power**
Other changes?

- There are many other changes since CHERI v9
- We’re trying to make adoption of CHERI-RISC-V easier

CSRks / SCRks for example:
- Removal of SCR space, and map them into the CSR space
- MTVEC (Legacy Mode) is MTVECC (Capability Mode) at the same address
- New CSRks like DDC are allocated into CSR space
- For a while we used aliased addresses but decided to lower the cost

Duplicate Mnemonics
- JALR/CJALR, LC/CLC etc.
- We removed the duplicates, to ease the burden on the toolchain and assembly writers.

Minimising ISA
- Removing instructions which can be emulated with small numbers of other instructions except where deemed performance critical

Easy Mode Switching
- Easily switch into one mode and back again with MODESW, which can be executed in the decoder.
- Previously it was necessary to install a new cap in PCC with the M-bit flipped
## What extensions do we have?

<table>
<thead>
<tr>
<th>Extension</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZcheriPurecap</td>
<td>Stable – bug fixes only</td>
<td>Base architecture for a CHERI purecap machine</td>
</tr>
<tr>
<td>ZcheriLegacy</td>
<td>Stable – bug fixes only</td>
<td>Implies ZcheriPureCap. Adds legacy RISC-V support</td>
</tr>
<tr>
<td>Zabhlrsc</td>
<td>Stable – bug fixes only</td>
<td>Byte/half LR/SC support (independent of CHERI)</td>
</tr>
<tr>
<td>Zstd</td>
<td>Software prototyping</td>
<td>Secure thread ID for compartmentalisation, the Thread ID is extended to capabilities with CHERI</td>
</tr>
<tr>
<td>ZcheriPTE</td>
<td>Research, in spec</td>
<td>Implies ZcheriPureCap. Revocation support by supporting cap accessed and dirty in page tables</td>
</tr>
<tr>
<td>ZcheriMultiLevel</td>
<td>Research, need PR</td>
<td>Support for locally/globally accessible capabilities with multiple levels</td>
</tr>
<tr>
<td>ZcheriTransitive</td>
<td>Research, need PR</td>
<td>Support for reducing capability permissions on loading</td>
</tr>
<tr>
<td>ZcheriTraceTag</td>
<td>Research, Need PR</td>
<td>Support for data capability trace with tags</td>
</tr>
<tr>
<td>ZcheriSanitary</td>
<td>Research, Need PR</td>
<td>Support for cleaning capabilities on compartment switching</td>
</tr>
<tr>
<td>ZcheriSystem</td>
<td>Research, Need PR</td>
<td>Support for exposing compartment IDs to the system (a better WorldGuard)</td>
</tr>
</tbody>
</table>
→ Getting to RVA23 Compatibility

• We’re currently building RVA22+CHERI, but we want to get to RVA23+CHERI, which has some challenges

• Vector+CHERI
  • Kind of simple – check every unmasked byte of every load/store against the bounds
    • The devil is in the detail though for complex sequenced load/stores
    • Consider adding VLC/VSC to load/store full vector registers including caps, and an associated VCMV to move a whole vector register to support Vector capability memcpy

• Vector+Hypervisor
  • We haven’t analysed this one yet, so have no comment on the implications
  • If you have ideas then please get in touch
Code Size Reduction

• Zcmt – table jump
  • Already in the CHERI-RISC-V spec

• Zcmp – push/pop
  • I think these are straight-forward but haven’t written up how to do them
  • The data-width doubles, and only RV32 is of interest, so effectively use the RV64 stack layout for RV32-CHERI-RISC-V
That’s all folks

- Collaborate with us: https://github.com/riscv/riscv-cheri
- Join the TG and the bi-weekly meeting
- Tell us what’s missing and help fill in the gaps
- Help drive CHERI to world domination