CX TG Requirements Discussion #1: CX State Model: One State or Multiple States?

Jan Gray, Acting Vice-Chair, Composable Custom Extensions TG

RVIA Disclosures

- https://wiki.riscv.org/display/HOME/Meeting+Disclosures

Our first requirements discussion breakout

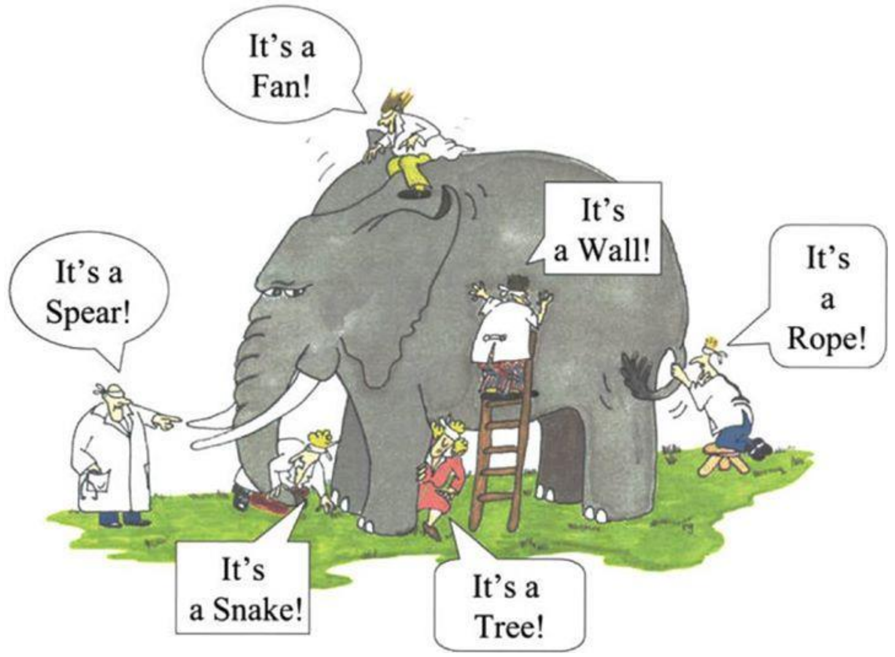
- 8:30: <= 5 min position statements, hold questions pls
- Open discussion
- 9:25: summing up
- Sharing our understanding, towards a consensus decision

- How to decide?
  - Thoroughly understand design, utility, and cost up and down the stack
  - Weigh additional spec, POC, and ratification time vs. decades of use
  - Weigh cost to change course later
- No decisions today

Jan’s position

- The CX software programming model’s CX state model must support
  - Composition of independent CX software libraries
  - Modern programming models, incl. tasks, coroutines, async methods
  - Efficiently and uniformly
- The CX state model should be flexible for diverse use cases
  - 0, 1, few, #harts, many × 0, S, M, L, XL CX state contexts
  - Isolated *and* shared state CX ISA design patterns
  - Bare metal and server class workloads and systems
- The basis spec’s ‘*m* harts : *n* CX state contexts’ (CX instances as hardware objects) is more novel, more complex, but more flexible, more performant, affordable, preferable

Let us be crisp about abstraction layers



- **User programming model**
- User ISA
- Supervisor OS concerns
- Supervisor ISA
- Machine ISA
- Logic interface

Avoid kicking the complexity can down the road

Focusing on CX Software Programming Model

Single or multiple = not that different

- Once on library init

```
cx = cx_open(MYCX_GUID, sharing);  // discover if CX present, allocate a state context
```

- Per client call

```
if (cx < 0) return sw(…);      // pure software if CX absent

cx_select(cx);                  // csrw cx_index,cx    // MYCX custom instructions
for (…)
  c[i] = cx_reg(FN,a[i],b[i]);  // custom-0 rd,FN,rs1,rs2
cx_select(0);                   // csrw cx_index,zero// built-in custom instructions
… = cx_status();                // csrr …,cx_status
```

Multiple instances of things?

- Programmers want multiple instances of resources
  - My library uses it, your library uses it too
  - My task/coroutine uses it, my other does too
- In software we create multiple ADTs or objects or resource handles
  - Singletons are awkward
- Same, if programming with accelerated software libraries?

Multiple instances of accelerators?

- In basis spec, built into the selector and CX mux'ing
  - cx_select(cx); // selects CX and CX's state context, one instruction
  - Optional

- In lieu of this: design CX state & behavior for shared single instance
  - No suspended stateful CX interactions: "get in, do it, and get out"
  - Repeatedly load CX context, compute, save CX context – not uniform
  - One CX state context but with ad hoc per-CX sub-context IDs
- Still need multiples, but we build them ad hoc

CX instances as hardware objects

- A uniform way to support multiple accelerator instances
- Key differences
  - Potential for CX software to open a second instance of a stateful CX
  - Programmer regards as an owned resource
  - Owned, not a consitutent of, hart’s architectural state
  - Like file descriptors, C++ objects, etc., not saved in makecontext()

Jan’s position

- The CX software programming model’s CX state model must support
  - Composition of independent CX software libraries
  - Modern programming models, incl. tasks, coroutines, async methods
  - Efficiently and uniformly
- The CX state model should be flexible for diverse use cases
  - 0, 1, few, #harts, many × 0, S, M, L, XL CX state contexts
  - Isolated *and* shared state CX ISA design patterns
  - Bare metal and server class workloads and systems
- The basis spec’s ‘*m* harts : *n* CX state contexts’ (CX instances as hardware objects) is more novel, more complex, more flexible, more performant, affordable, preferable

Other positions

BACKUP

What is this TG about?

- Composition is our paramount concern
  - Prize #1: a composable extensions ecosystem with a catalog of mix-and-match reusable components
  - CX software libraries that issue CX custom instructions to CXUs
  - "What works separately, works together"

- Uniformity is very important too
  - Prize #2: a good enough, common way to do common CX stuff
    - So you don't have to reinvent everything
    - So my way is also your way, so our ways are compatible
  - Here, uniform state model, OS support, and programming model

Some other design tenets [basis]

- Diversity
  - HW: simple … complex CPUs, CXs/CXUs, topologies
    - E.g., k clusters × { m harts × n CXs × [s_0,…,s_{n-1}] CX state contexts }
  - SW: bare metal … RTOS … OS … HV+OSs+VMs/JITs+apps
  - One trusted org @ one time … dozens of random orgs @ many years
- Longevity
  - (Here) support diverse programming models, legacy, modern, emerging
- Simple, frugal, fast

Stateless and stateful CX custom instructions [basis]

- Basis spec requires & achieves invariance under composition
- Stateless CX: “each [insn] is a pure function of its operands”
- Stateful CX: “each [insn] may access, and as a side effect, update, the hart’s current state context of the extension (only)”
  - “The behavior … only depends upon the *series* of [insns] …”
  - “Besides updating extension state, …, and a destination register, an [insn] has no effect upon any other state or behavior of the system”

- TG: revisit in “composability criteria”

Most CXs are stateful [basis] *(§1.3.1)*

- CX custom instructions may be stateful
  - May access accumulators, registers, register files, ..
  - Conflict-free CX custom CSRs
- Uniform CX state context management
  - Supports 100s of CX state contexts, per CX, per system
  - Any mapping from harts to CX state contexts, 1:1, 1:*n*, *n*:1
  - Uniform OS access control, virtualization, context switching of any CX state context