Design and Rationale of the Draft Proposed RISC-V Composable Custom Extensions Spec

v.2023-08-06

Jan Gray
Gray Research LLC
jan@fpga.org

SPEC: https://github.com/grayresearch/CX/

2019-2022 design contributors

- Tim Ansell, Tim Callahan, Jan Gray, Karol Gugala, Olof Kindgren, Maciej Kurc, Guy Lemieux, Charles Pappon, Tim Vogt

Introduction *(§1)*

RISC-V custom extensions' reuse problem *(§1)*

- Standard extensions layer and compose. But take years to ratify
- Custom extensions: rapid in-house domain optimized solutions
- … with incompatible opcodes, discovery, computation, state, error handling, tools, versioning, hardware signaling
  - *RoCC, CV-X-IF, PCPI, SCAIE-V, MCUX, ACE, CODA*
- Solution silos → fragmentation

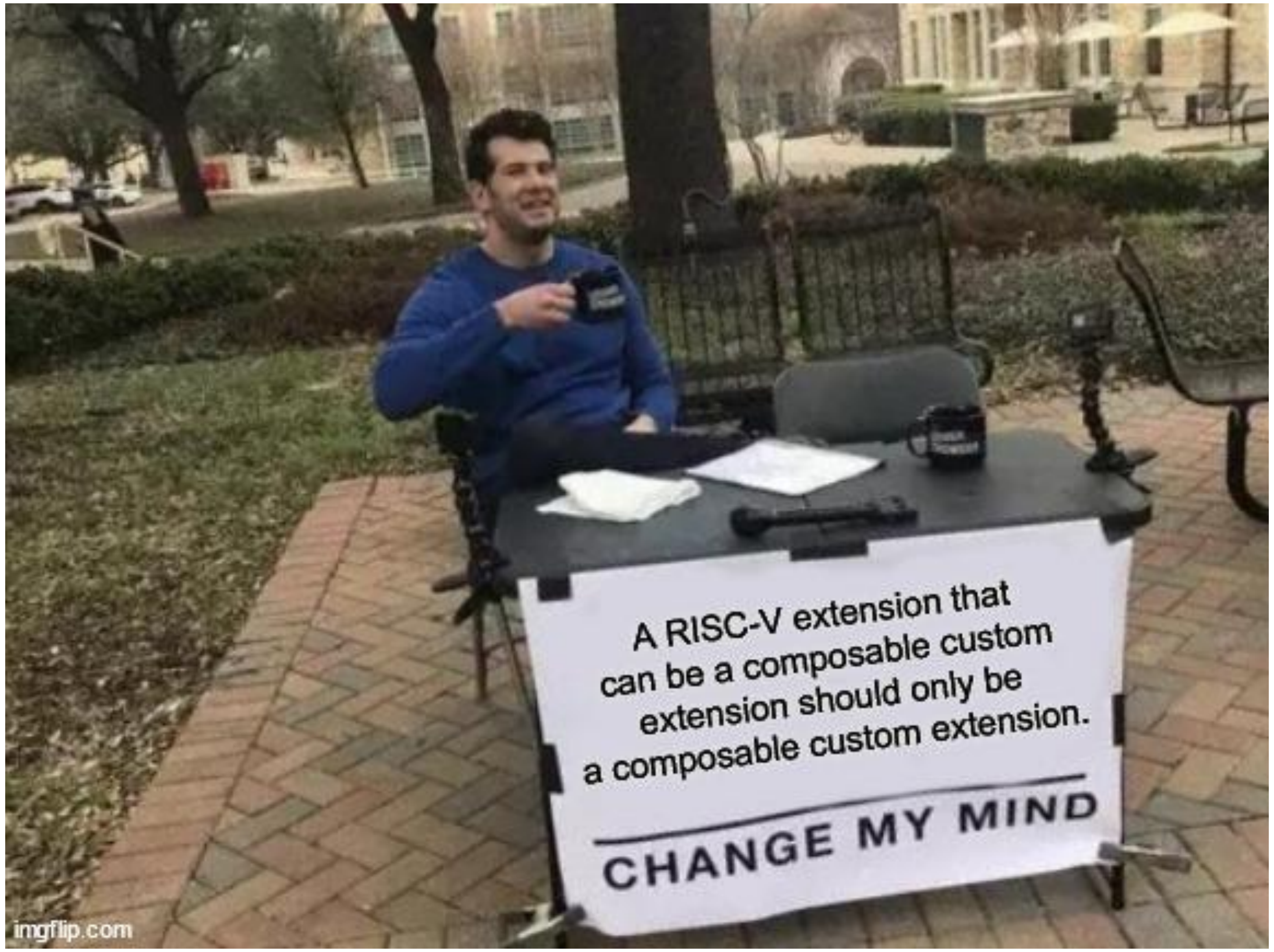
A mix & match **composable** custom extensions ecosystem *(§1.1)*

- *Agility* of custom extensions **+** *composability* of standard extensions
- Adopt some SW and HW interop standards so extensions *coexist*

- Then *anyone* can define, develop, and/or use:
  - A new extension, its software libraries, its hardware implementations
  - A processor that uses any of these, a system composed of these
  - Tools for same

Keeping RISC-V simple, open, agile

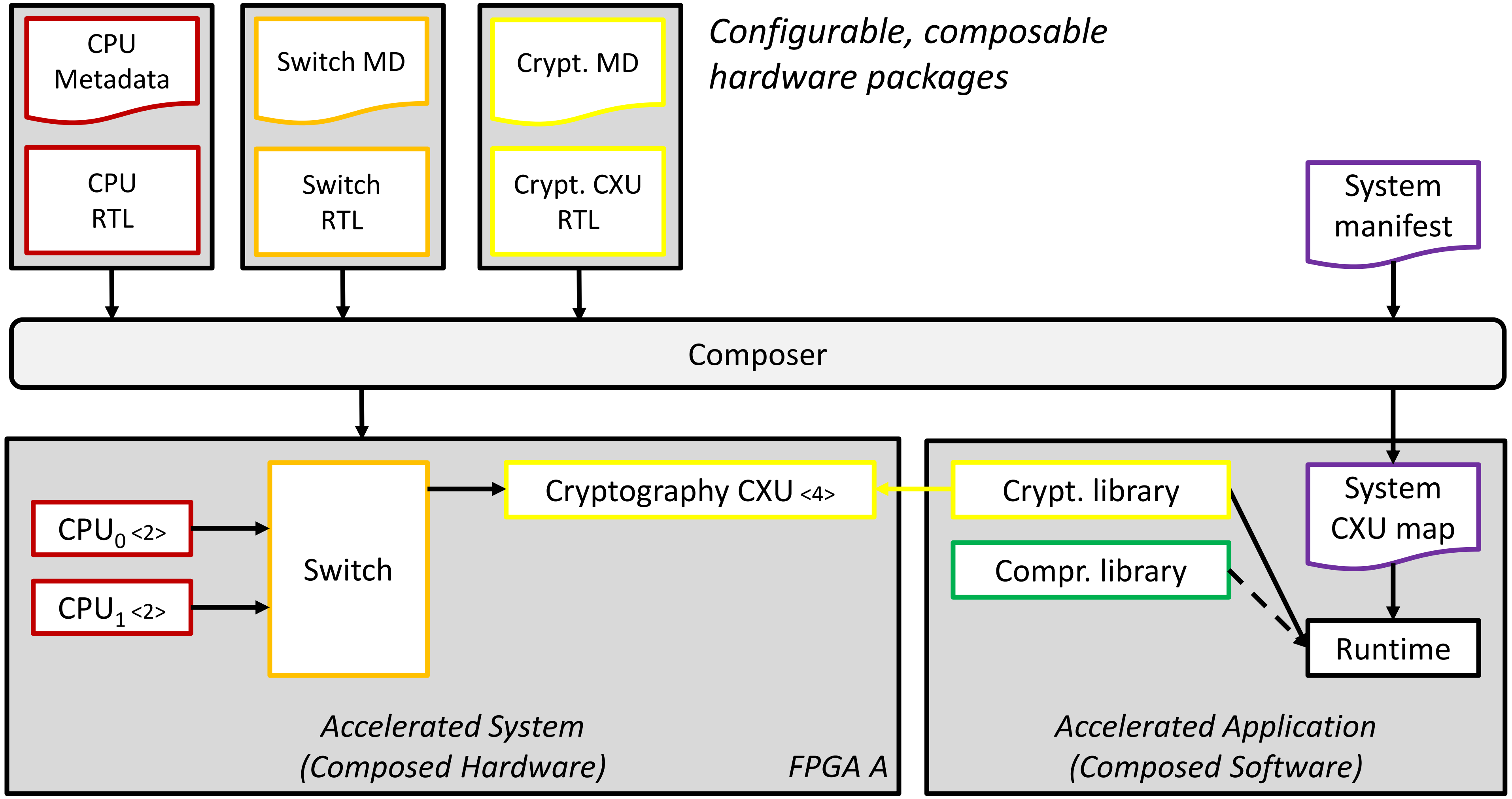
- Less pressure to add domain-specific standard extensions


A RISC-V extension that can be a composable custom extension should only be a composable custom extension.
CHANGE MY MIND
imgflip.com

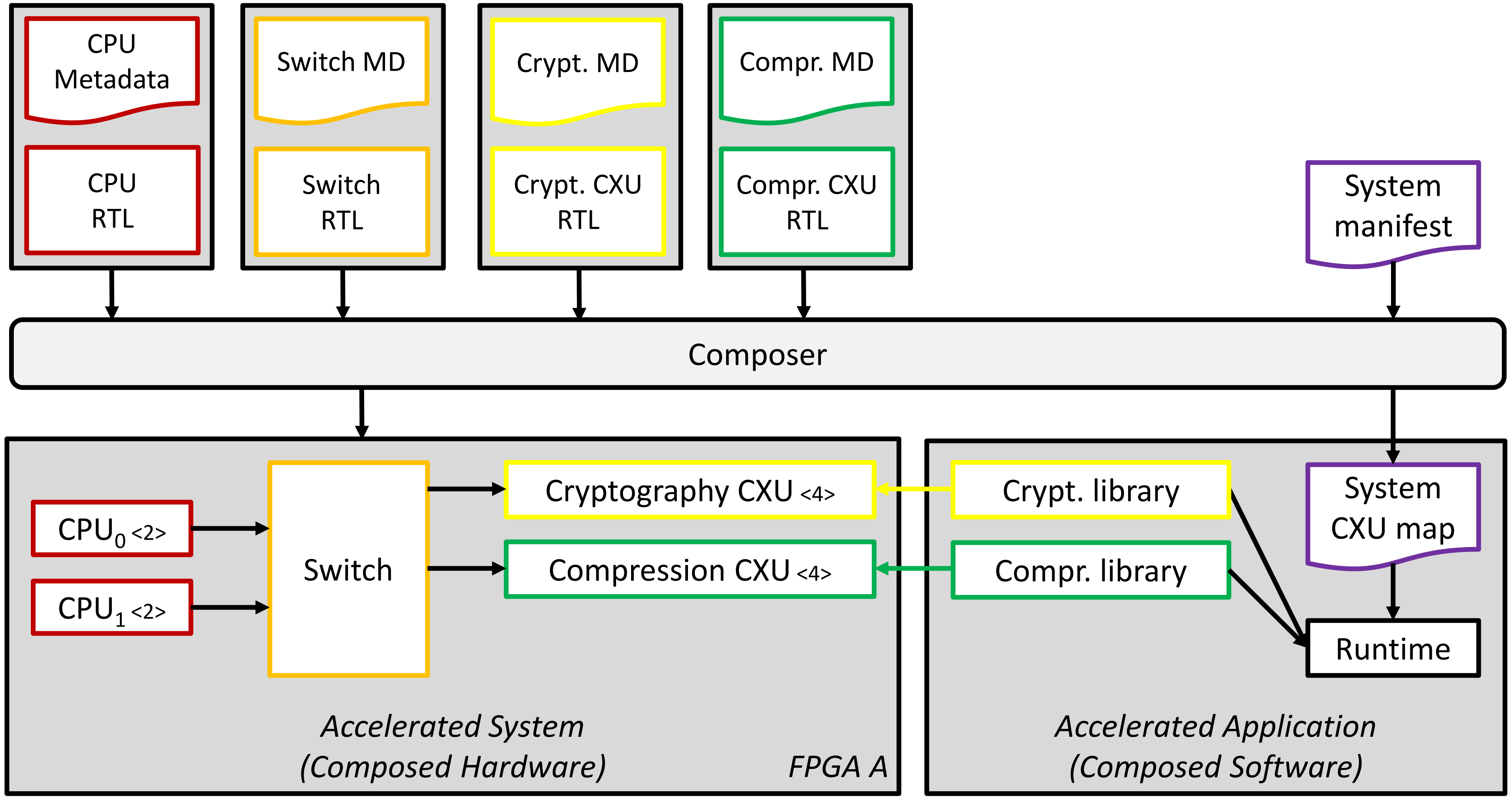
Key elements

- **Composable extension (CX)**: a *composable* custom extension’s interface contract
- **Composable extension unit (CXU)**: a hardware core that implements a CX
- **Composable extension library (CX lib)**: software that selects a CX, issues its custom instructions

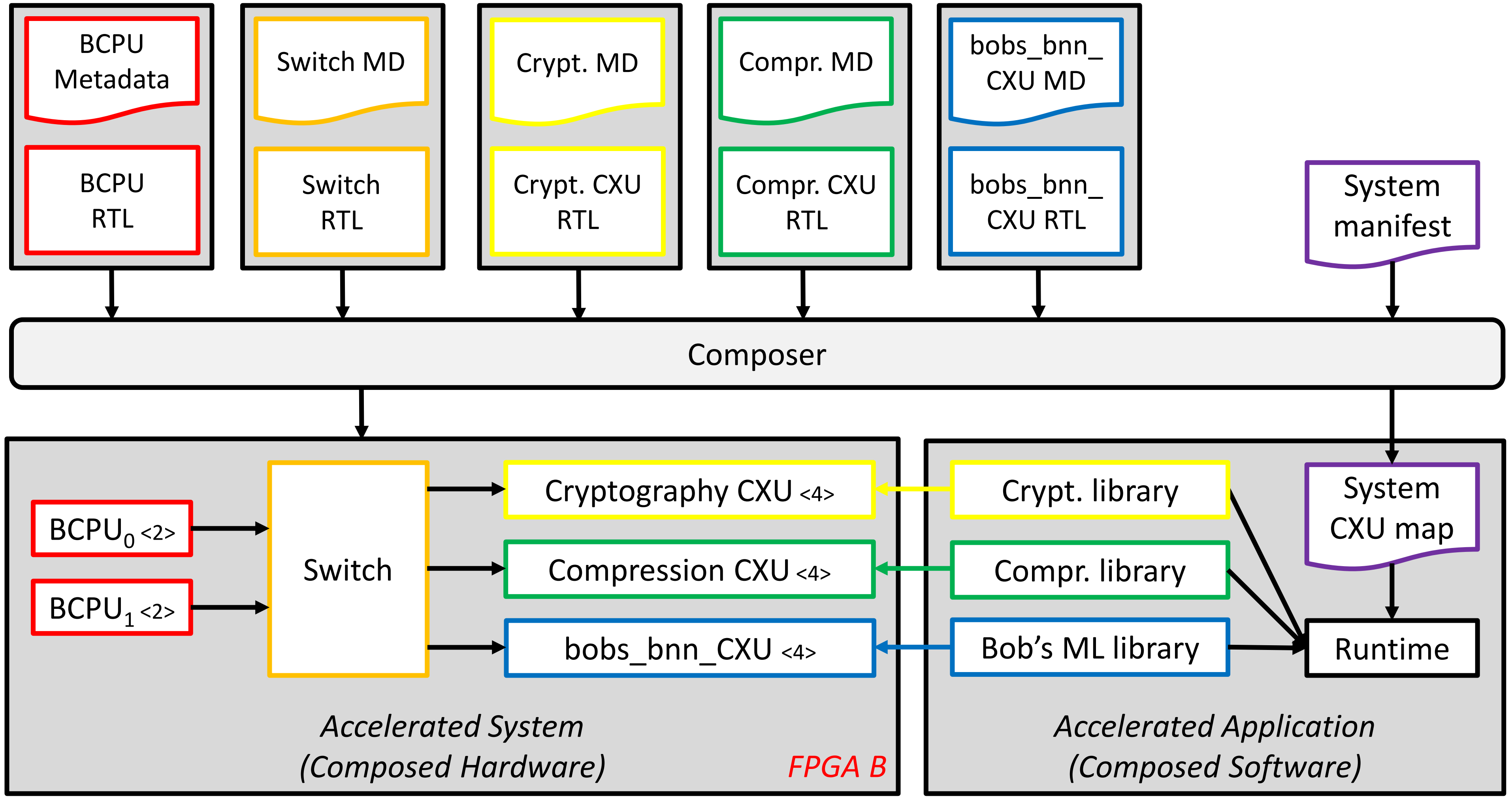
Example: Alice's SoC *(§1.2)*



Alice’s SoC **v2**



Bob’s SoC


BCPU Metadata
BCPU RTL
Switch MD
Switch RTL
Crypt. MD
Crypt. CXU RTL
Compr. MD
Compr. CXU RTL
bobs_bnn_ CXU MD
bobs_bnn_ CXU RTL
System manifest
Composer
BCPU$_0$ <2>
BCPU$_1$ <2>
Switch
Cryptography CXU <4>
Compression CXU <4>
bobs_bnn_CXU <4>
Crypt. library
Compr. library
Bob’s ML library
System CXU map
Runtime
Accelerated System (Composed Hardware)
FPGA B
Accelerated Application (Composed Software)

Objectives ⇨ design rationale

- Easy, robust, routine, boring **composition**
  - Across orgs, decades, platforms, tools
- Decentralized
- Diversity
- Open, nonproprietary, free, limitless
- Simple, frugal, fast
- Stable binaries, stable RTL
- Secure (privileged access control)
- Promote uniformity …

Promote uniformity

- Uniform naming (IDs), discovery, versioning
- Uniform, unlimited, conflict-free instruction encodings
- Uniform stateful extensions & uniform (no code) context save/restore
- Uniform error signaling
- Uniform access control
- Uniform logic interfaces
- Uniform metadata

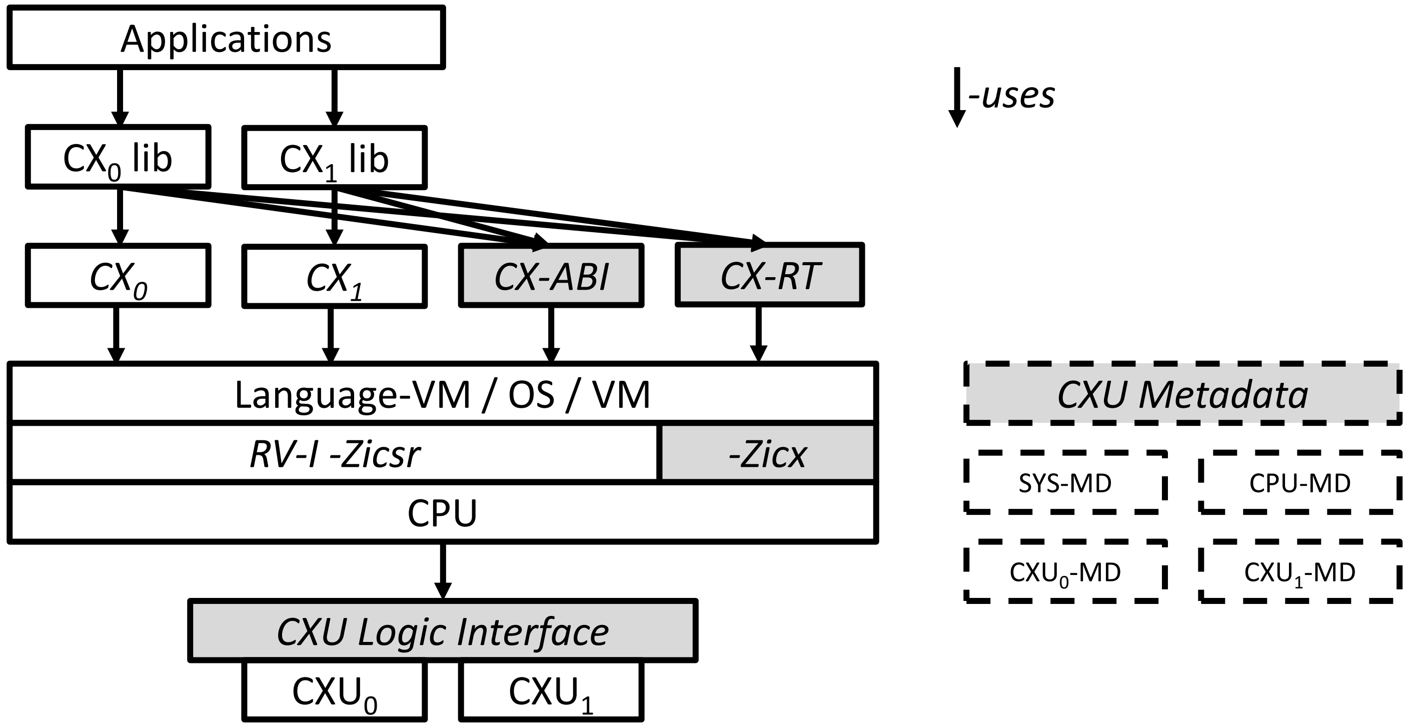
- *One **good enough** way to do things vs. many incompatible ways*

New interop standards

- **ISA:** HW-SW: composable extension multiplexing: -Zicx
  - A standard ABI for composition of CXs and CX libraries
  - Software first selects hart's current CX & CX-state
  - Each CX enjoys full custom instruction encoding space
- **Non-ISA:** HW-HW: CXU Logic Interface (CXU-LI)
  - A signaling standard for composition of { CPUs + CXUs } complexes
- *Better together*

Standard interfaces and formats *(§1.4)*

- CX Runtime: library services (discovery, state management, …)
- **-Zicx**: “composable extensions” extension, new CSRs for CX multiplexing
- CXU Logic Interface & metadata



Scope: what *kinds* of custom extensions? *(§1.3)*

- Invariant under composition ⇨ *isolated ones*
- V1: *custom function* instructions: int registers & an isolated state context
  - None of: float regs, vector regs, memory, CSRs, PC, exceptions

- *Cf.* RoCC, CV-X-IF, PCPI, SCAIE-V, MCUX, ACE, CODAL, …
  ⇨ *another presentation*

Stateful composable extensions *(§1.3.1)*

- Custom function instructions may R/W current state context data, including private RAM and registers
- Each CX/CXU configured with *n* isolated state contexts

- **IStateContext** (§1.4.4) standard CF instructions for uniform access to state

```
interface IStateContext {
//  CF_ID       custom function
    [1023] int  cf_read_status ();
    [1022] void cf_write_status(int status);
    [1021] int  cf_read_state  (int index);
    [1020] void cf_write_state (int index, int state);
};
```

CX (V1) scenarios: more than meets the eye

- Stateless
  - Bitmanip: X[rd] = *f*(cf, X[rs1], X[rs2]/imm) – all of Zba Zbb Zbc Zbs
- **Stateful**
  - CSRs: X[rd] = f(cf_write_state, X[rs1], X[rs2]) – CX csrrw
  - Reduce: (X[rd],**A**) = *f*(cf, **A**, X[rs1], X[rs2]) – dot product, multiprec math
  - Vector: **V**[rd] = *f*(cf, **V**[rd], **V**[rs1], **V**[rs2]) – vector-regfile state
    {X,**V**}[rd] = *f*(cf, **V**[rs1], X[rs2]) – put/get elements
  - 4r1i: (**i,a,b**) = ({cf_0,*custom*}, X[$rs1_0$], X[$rs2_0$]) – 2 instruction sequence
    X[rd] = *f*(cf_1, X[$rs1_1$], X[$rs2_1$], **i**, **a, b**)
  - Channels: X[rd] = *putget*(**chan**[cf.chan], cf.{put,put2,wait}, X[rs1], X[rs2])
  - f(RAM): put-**data***; operate(**data**)***;** get-**data*** – crypto, ML, …
  - Async: **FSM** = f(cf_0, …); . . . . .; X[rd_1] = f(cf_1, **FSM**, X[rd_0], …)
- CPU support for CX V1 enables so much for so little

Scope: V2 ideas, directions

- Compose V1 with PULP stream semantic registers
- Access memory: CXU⇨CPU; precisely ordered; disjoint/isolated regions
- Enable custom instructions beyond custom-[012] opcodes
- Register pairs e.g. (X[rd],X[rd+1])
- Speculative issue / cancelation of stateful custom instructions

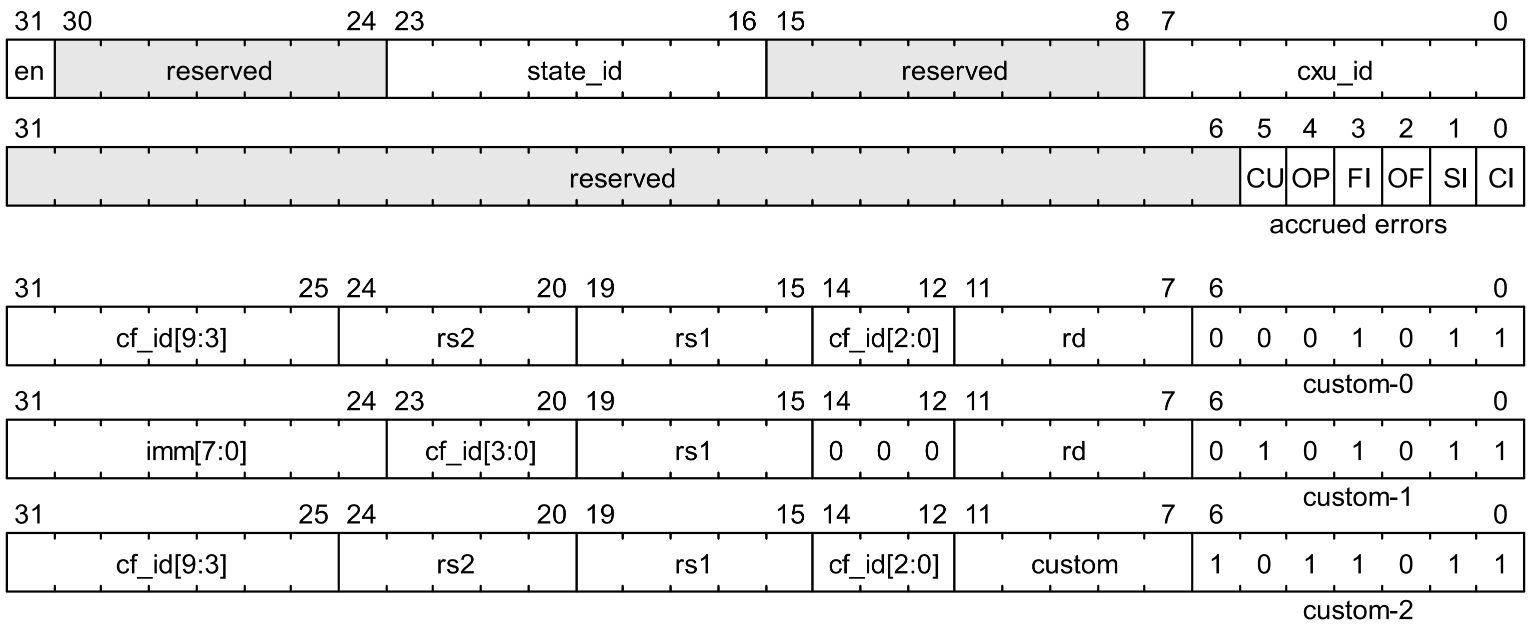
Composable extension multiplexing

Collision-free custom instructions

- **mcx_selector** ::= hart's *current* (CX/CXU, state)
- CX lib: **csrw mcx_selector**; then **custom-[012];*** issues requests to *the* CXU
- CXU performs each request, updating its state; write dest reg & **cx_status**

```
mcx_selector CSR
cx_status CSR

cx_reg  cf_id,rd,rs1,rs2
cx_imm  cf_id,rd,rs1,imm
cx_flex cf_id,rs1,rs2
```



31 30 24 23 16 15 8 7 0
en | reserved | state_id | reserved | cxu_id
31 6 5 4 3 2 1 0
reserved | CU | OP | FI | OF | SI | CI
accrued errors
31 25 24 20 19 15 14 12 11 7 6 0
cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | rd | 0 0 0 1 0 1 1
custom-0
31 24 23 20 19 15 14 12 11 7 6 0
imm[7:0] | cf_id[3:0] | rs1 | 0 0 0 | rd | 0 1 0 1 0 1 1
custom-1
31 25 24 20 19 15 14 12 11 7 6 0
cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | custom | 1 0 1 1 0 1 1
custom-2

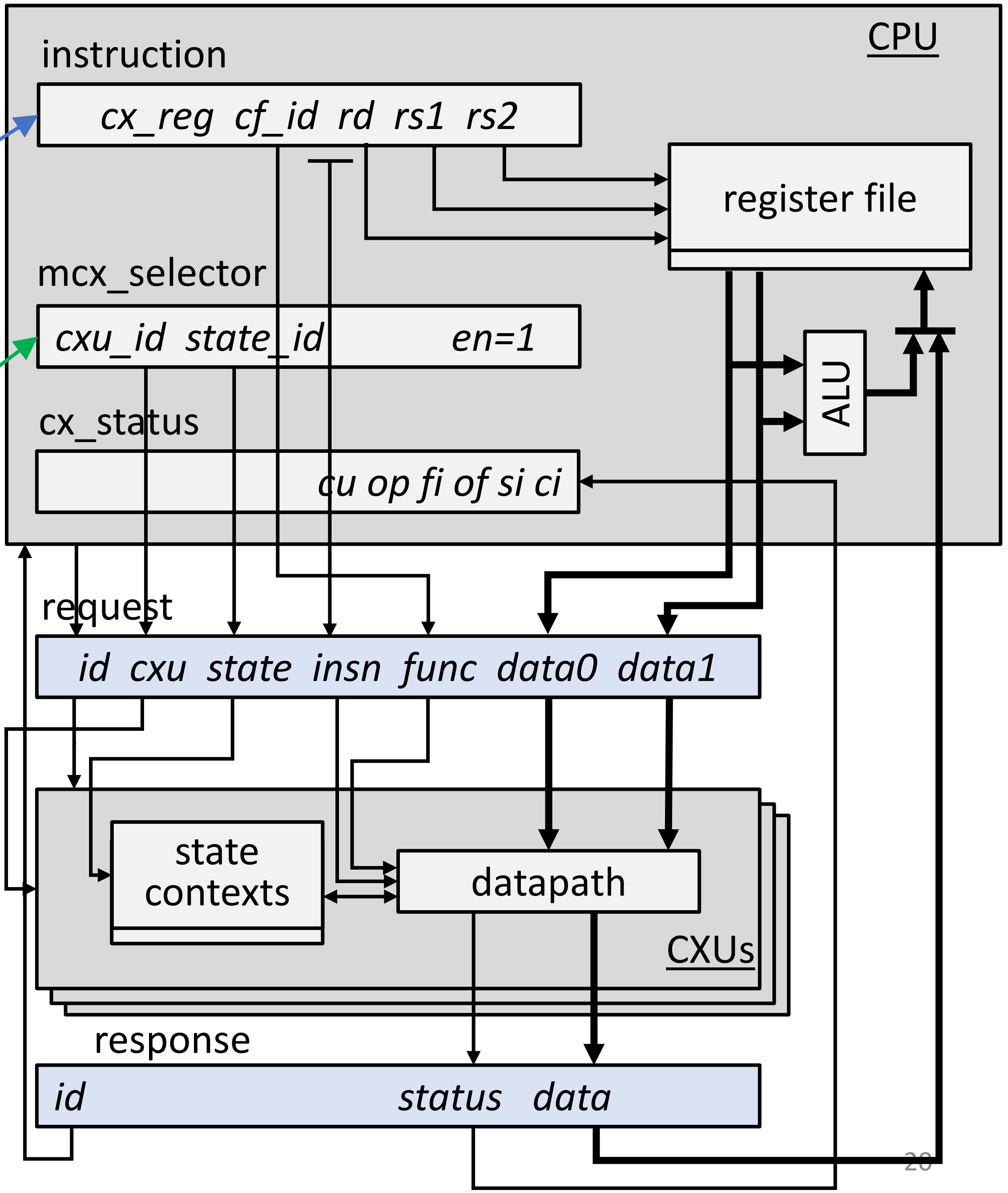
Multiplexing in detail

The SW ⇔ HW interface

```
csrw   mcx_selector,x1
cx_reg 101,x4,x2,x3
cx_reg 102,x7,x5,x6
…

csrw   mcx_selector,x10
```
cx_reg 11,x14,x12,x13
```
cx_reg 12,x17,x15,x16

…
```



CX Runtime and CX ABI *(§1.4.5)*

- CX libraries use CX Runtime services
  - Discovery: map CX_ID (GUID) ⇨ CXU_ID, if present
  - State: alloc/free, save/reload a state context
  - Change CX selectors
  - Access control
  - *Runtime services, in software, keeps hardware frugal*
- CX ABI
  - -V **setvl** analogy: **csrrw mcx_selector** prefixes groups of custom instructions
  - mcx_selector: callee save, restore on stack unwinds
  - Basic: scoped by C++ RAII object constructor/destructor
  - Better: compiler inserts and optimizes mcx_selector writes

Example CX programming model

C++ RAII object to scope the custom instructions

- Try to select the IBitmanip CX, issue its custom instructions, deselect it

```
if (CX cx(IBitmanip_CX_ID); cx) // CX():  csrrw x1,mcx_selector,x2
  count = cf(pcnt, data, 0);    //        cx_reg pcnt,rd,data,x0
                                // ~CX(): csrw mcx_selector,x1
else
  count = popcount(data);       // no CX/CXU: use software
```

- *Better: retain selector values*

Versioning story *(§1.6)*

- Over decades, change happens (AVX, Direct3D12)
- A CX is a named, *immutable* interface contract
- Any change is a new CX, so you mint a new CX_ID
- A CXU may implement >1 CX, e.g., old and new CX contracts

- Negotiation: CX lib asks CX Runtime for CX_2 else CX_1 else fails

Inspired by Microsoft Component Object Model *(§1.9)*

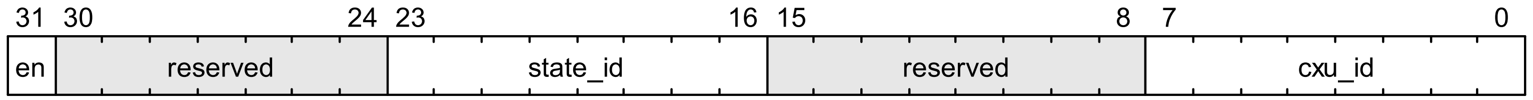
- Framework for composition of 100s of diverse software components
- Proven at scale for three decades across Windows, Office, DirectX
- In common with CX proposal
  - Immutable interface contracts, self-named by GUIDs ✓
  - Components implement 1+ interfaces ✓
  - Separately authored, separately versioned, substitutable ✓
  - Dynamic discovery affords best-interface negotiation ✓

Composable Extensions / -Zicx *(§2)*

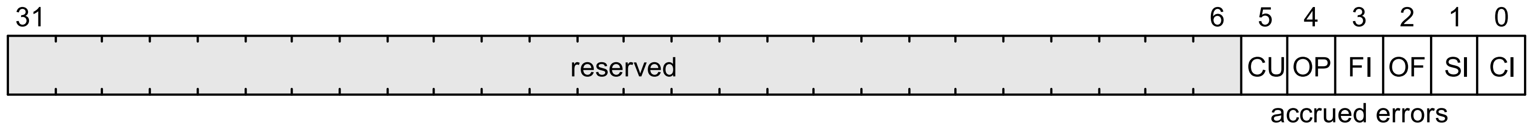
Four new CSRs *(§2.2)*

- M-mode composable extension multiplexing

mcx_selector

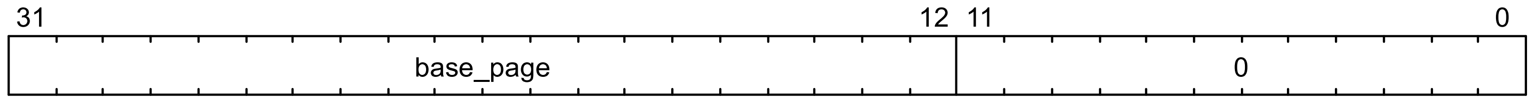


cx_status

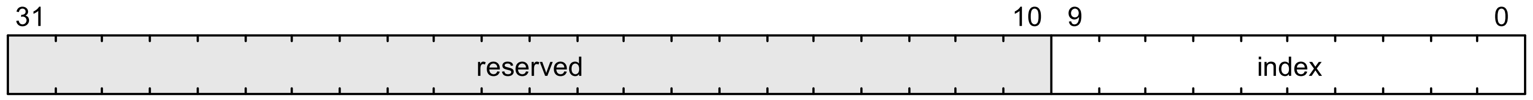


- Access controlled custom extension multiplexing

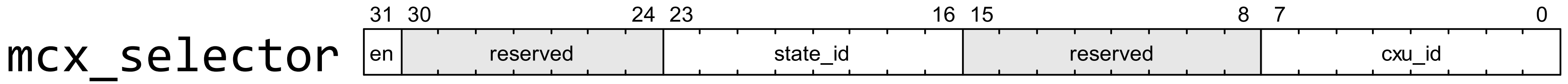
mcx_table



cx_index



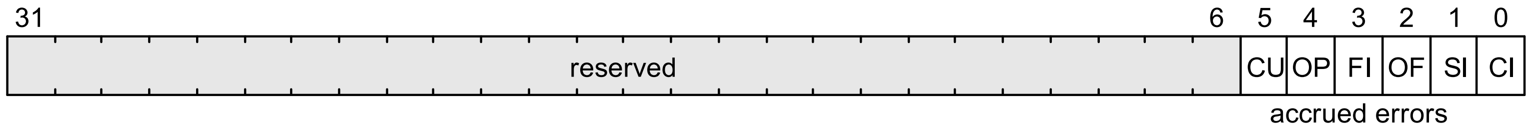
mcx_selector CSR



- .en=1 enables composable extension multiplexing
- .en=0 custom-0/1/2 does whatever it already does on this CPU
- .cxu_id selects current configured CXU
- .state_id selects current state context of that CXU
- Only accessible in machine mode

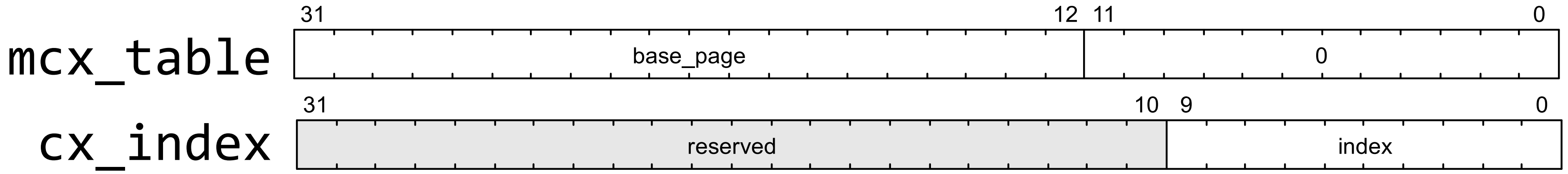
cx_status CSR

`cx_status`



- Accrues CX/CXU error status flags (CXs don't raise exceptions, *cf.* **fcsr**)
- R/W in all priv modes
- Set when mcx_selector.en=1 and a custom-[012] instruction error
  - .CI: invalid cxu_id
  - .SI: invalid state_id
  - .OF: that state context is OFF
  - .FI: no such custom function
  - .OP: operation error, *e.g.,* divide by zero
  - .CU: (stateful CX) custom error: errno etc. via IStateContext::cf_read_status/state

Access controlled CX multiplexing CSRs *(§2.7)*



- Grant/deny unpriv software access to CXs and CX state
  - + Deny inferring CX use in other harts – deny certain side channel attacks
- Unpriv: change CX selector with one CSR write, no OS detour
  - Priv OS: at ***mcx_table** keep hart's 4 KB = 1024 CX selector table, unpriv inaccessible
  - Priv OS: provides opaque CX selector indices to unpriv user code
  - Unpriv: **csrrw** old_index=x1, **cx_index**, x2=new_index
  - CPU: mcx_selector = mcx_table[cx_index] *"at next priv mode"*
- Prefer cx_index writes to mcx_selector writes

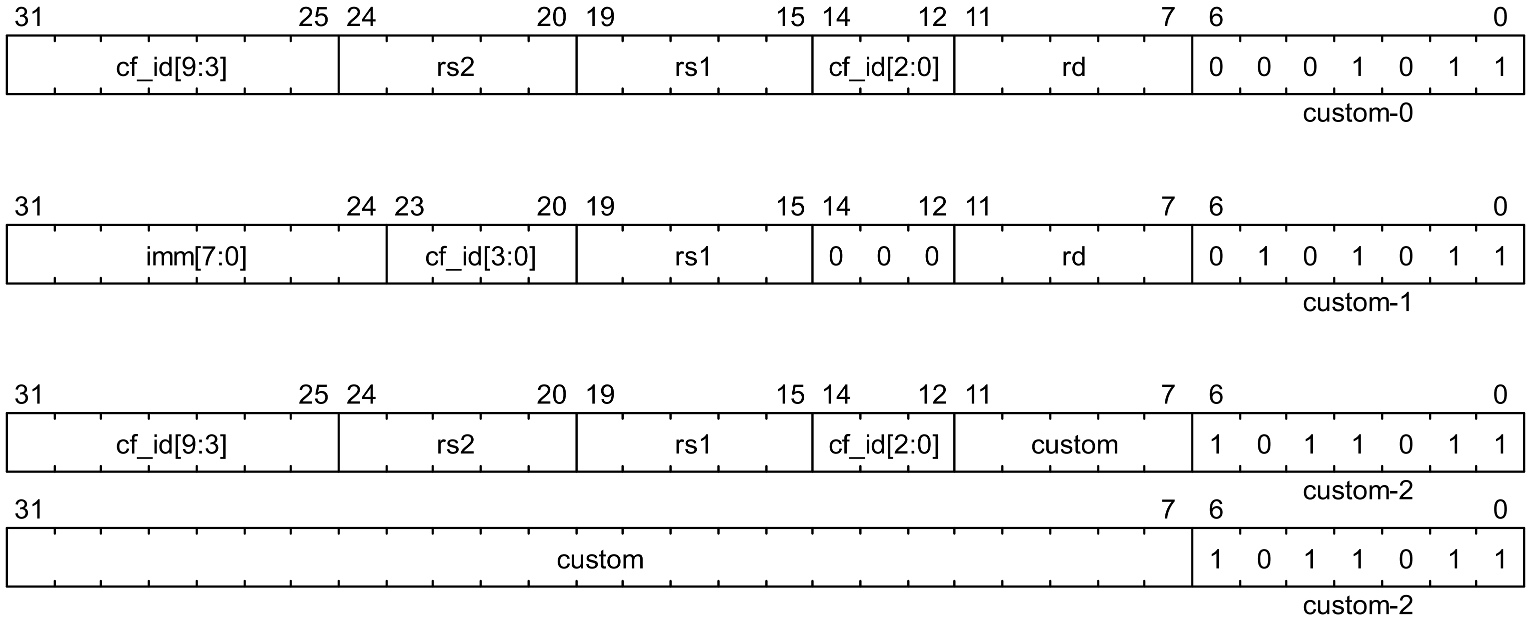
CX custom function instruction encodings *(§2.3)*

`cx_reg  cf_id,rd,rs1,rs2`

`cx_imm  cf_id,rd,rs1,imm`

`cx_flex cf_id,rs1,rs2`

`(cx_flex25 custom)`


31 25 24 20 19 15 14 12 11 7 6 0
cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | rd | 0 0 0 1 0 1 1
custom-0
31 24 23 20 19 15 14 12 11 7 6 0
imm[7:0] | cf_id[3:0] | rs1 | 0 0 0 | rd | 0 1 0 1 0 1 1
custom-1
31 25 24 20 19 15 14 12 11 7 6 0
cf_id[9:3] | rs2 | rs1 | cf_id[2:0] | custom | 1 0 1 1 0 1 1
custom-2
31 7 6 0
custom | 1 0 1 1 0 1 1
custom-2

- Ripe for discussion / redesign
- Future proofing? Anticipate V2+ needs

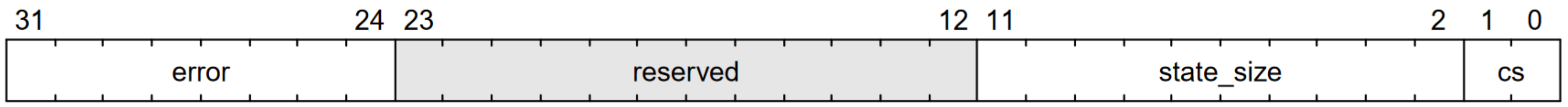
Implicit fences *(§2.2.5)*, precise exceptions *(§2.3)*

- Implicit fence between CX CSR access and any custom-[012] instructions
  - CX CSR access *“happens-before”* custom instructions*
  - custom instructions* *“happens-before”* CX CSR access
- *E.g.,* **csrr cx_status** awaits responses of prior custom instructions
- Precise exceptions vs. speculative issue of stateful custom instructions

IStateContext

Custom function	CF_ID	Assembly instruction	Encoding
cf_read_status	1023	cx_read_status rd	cx_reg 1023,rd,x0,x0
cf_write_status	1022	cx_write_status rs1	cx_reg 1022,x0,rs1,x0
cf_read_state	1021	cx_read_state rd,rs1	cx_reg 1021,rd,rs1,x0
cf_write_state	1020	cx_write_state rs1,rs2	cx_reg 1020,x0,rs1,rs2

- Each stateful extension must include these CF instructions
- **cf_{read,write}_status** accesses **state context status** word:



  - .cs (like **mstatus.XS**): 0:off 1:initial 2:clean 3:dirty
- **cf_{read,write}_state** to save/restore raw state context data

Composable Extension Unit Logic Interface (CXU-LI) *(§3)*

CXU-LI objectives ⇨ design rationale

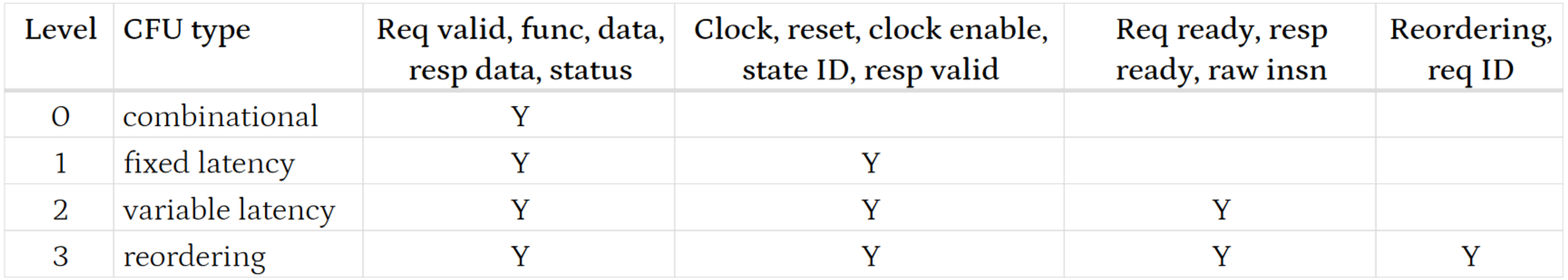
- Modular, platform neutral, composable extensions' implementations
- Compose CPUs+CXUs without changing CPU+CXU RTL
- Handle diverse CPUs (pipelines) and CXUs
- LUT frugal
- Promote uniform signaling, naming, configuration, validation, metadata

Zen of CXUs

- A CXU implements one (or more) composable extensions
- CPU `custom-*`$^+$ ⇨ CXU requests ⇨ CXU ⇨ CXU responses ⇨ CPU
- Configurable width, latency, state, … – described in metadata
- CXU Logic Interface feature levels
  - 0: combinational; 1: fixed latency; 2: variable latency; 3: reordering
  - Each CXU and CPU speaks a specific CXU-LI level
- Composition: CPUs ⬄ switches & adapters ⬄ CXUs

CXU-LI feature levels *(§3.3)*

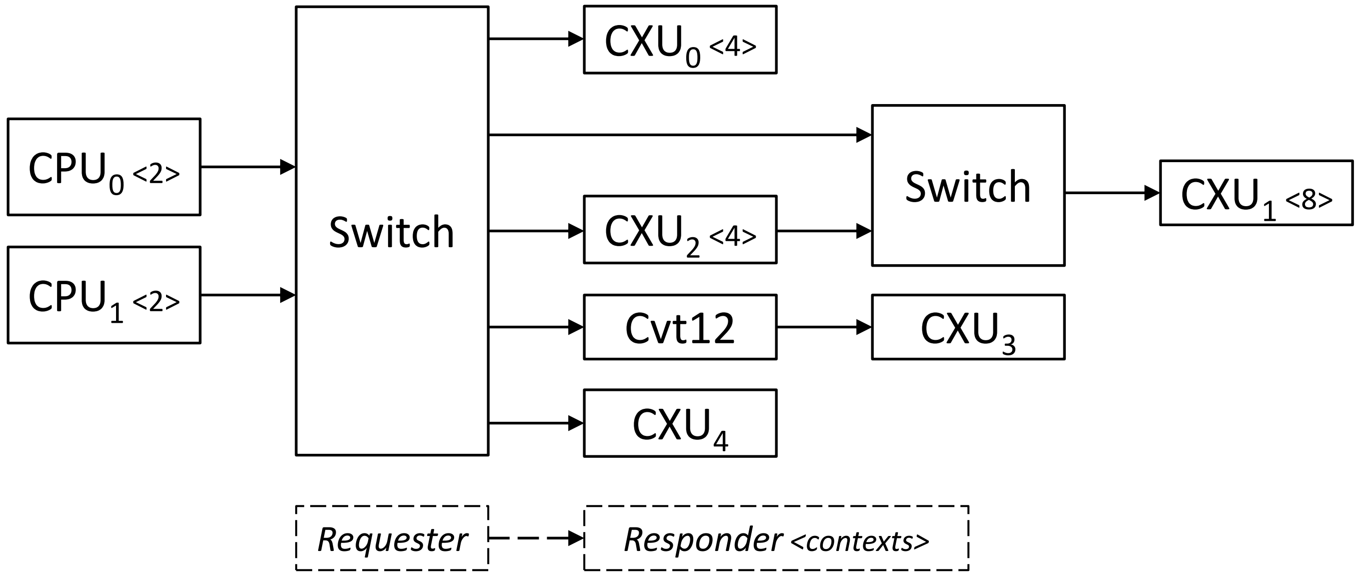
- Keep simple use cases simple and frugal, make complex use cases possible
- Accommodate simple or complex CPUs, simple or complex extension



Level	CFU type	Req valid, func, data, resp data, status	Clock, reset, clock enable, state ID, resp valid	Req ready, resp ready, raw insn	Reordering, req ID
0	combinational	Y			
1	fixed latency	Y	Y		
2	variable latency	Y	Y	Y	
3	reordering	Y	Y	Y	Y

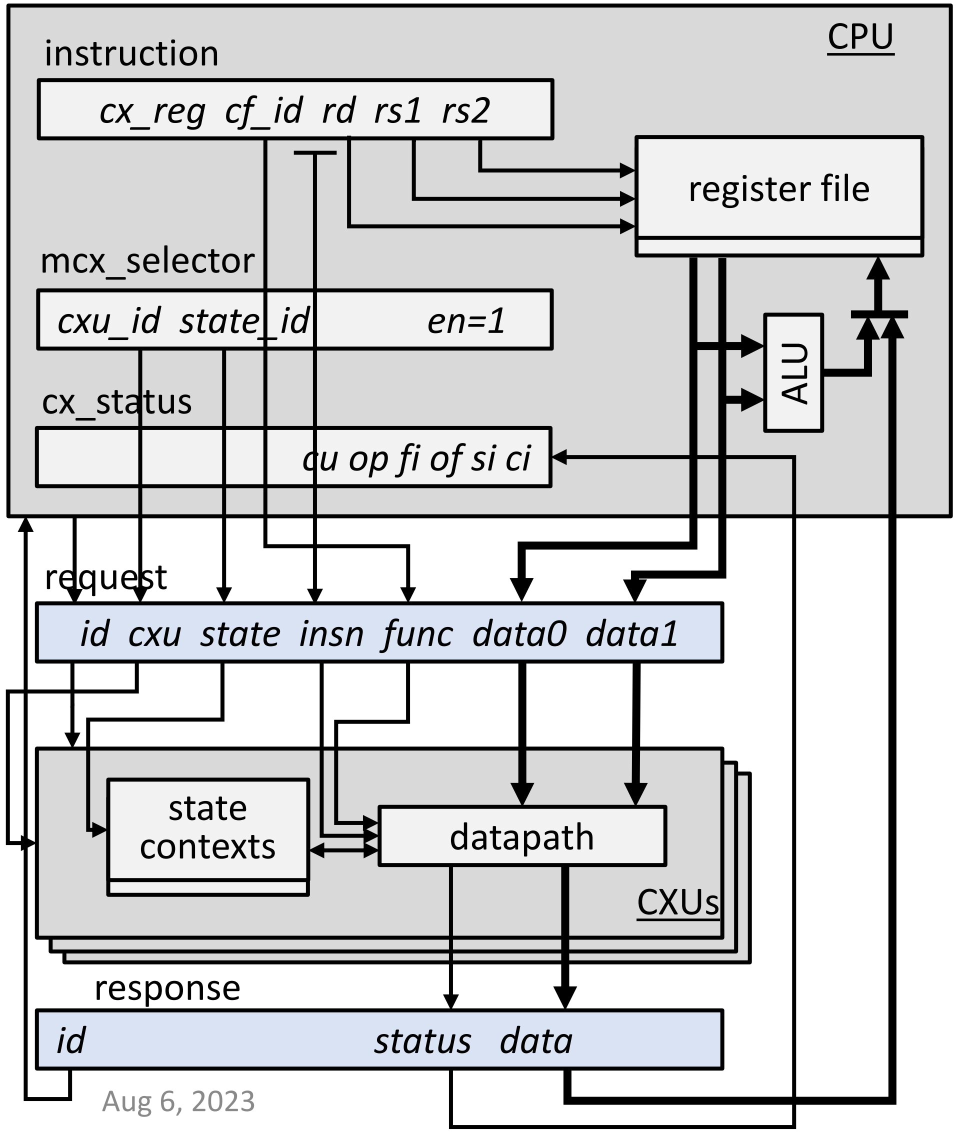
CPUs × CXUs composition *(§3.2)*

- CPUs ⬄ switches & adapters ⬄ CXUs
- Powered by CPU & CXU packages' metadata
- CPUs *and CXUs* can request CXUs; CXUs form a DAG



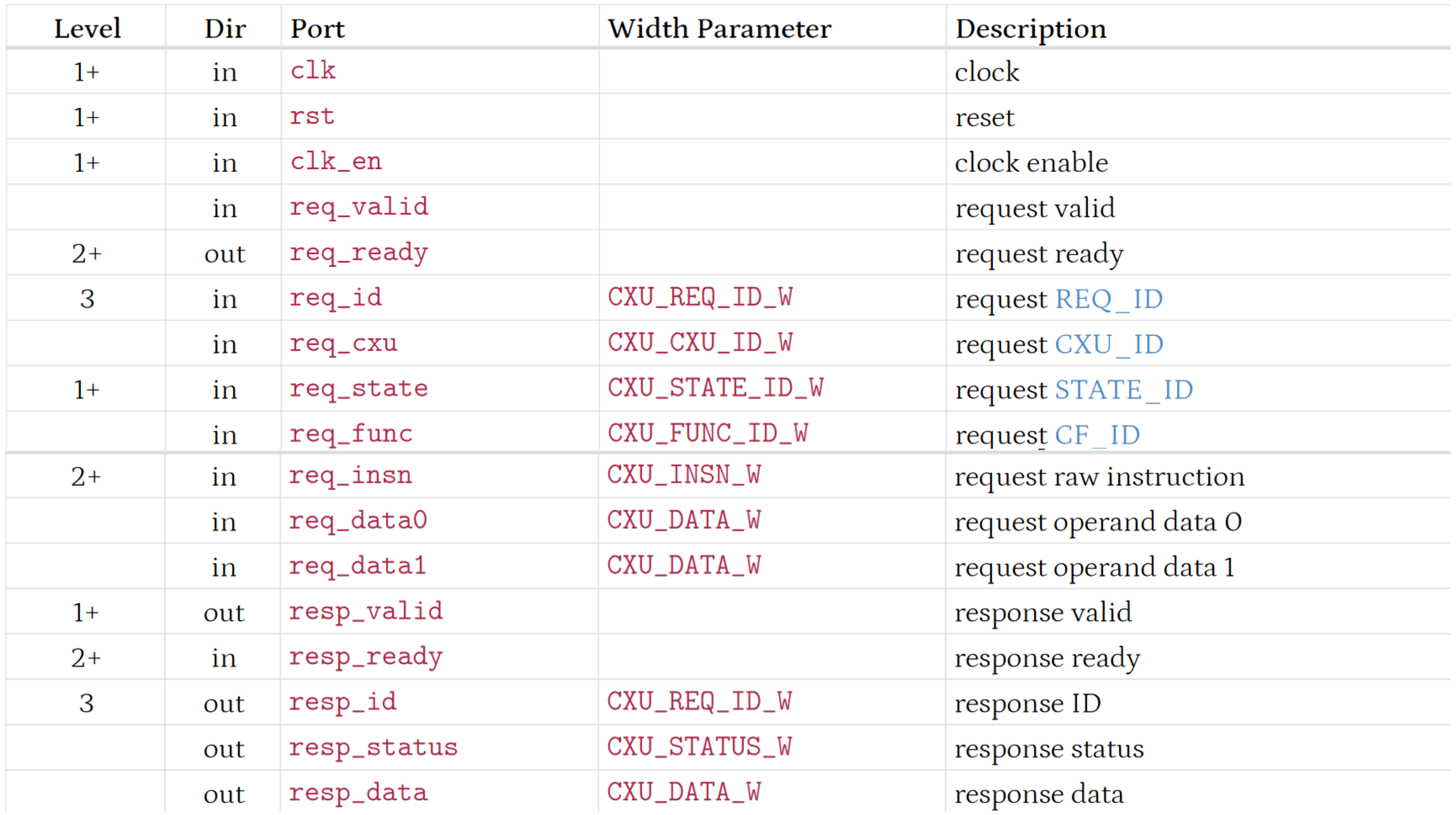
- CXUs composition = configuration constraint satisfaction, else infeasible
  - *E.g.:* CXU-L1 CPUs + CXUs: find a feasible system-wide CXU_LATENCY

CXU-LI HW-SW interface, again



CXU-LI signal	← Source or → Destination
req_id	← CPU
req_cxu	← mcx_selector.cxu_id
req_state	← mcx_selector.state_id
req_insn	← insn
req_func	← insn.cf_id
req_data0	← R[insn.rs1]
req_data1	← R[insn.rs2] {custom-0/-2} or insn.imm {custom-1}
resp_id	→ CPU
resp_status	→ cx_status bits
resp_data	→ R[insn.rd] {custom-0/-1}

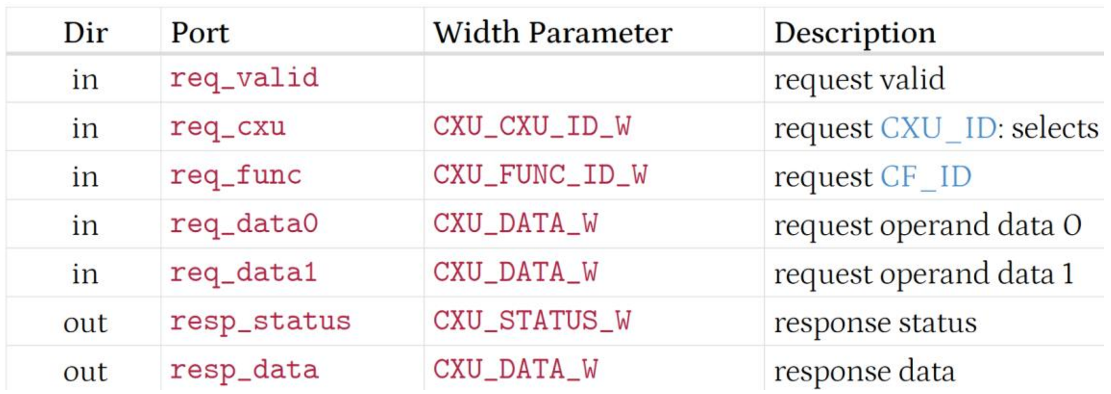
CXU-LI port signaling



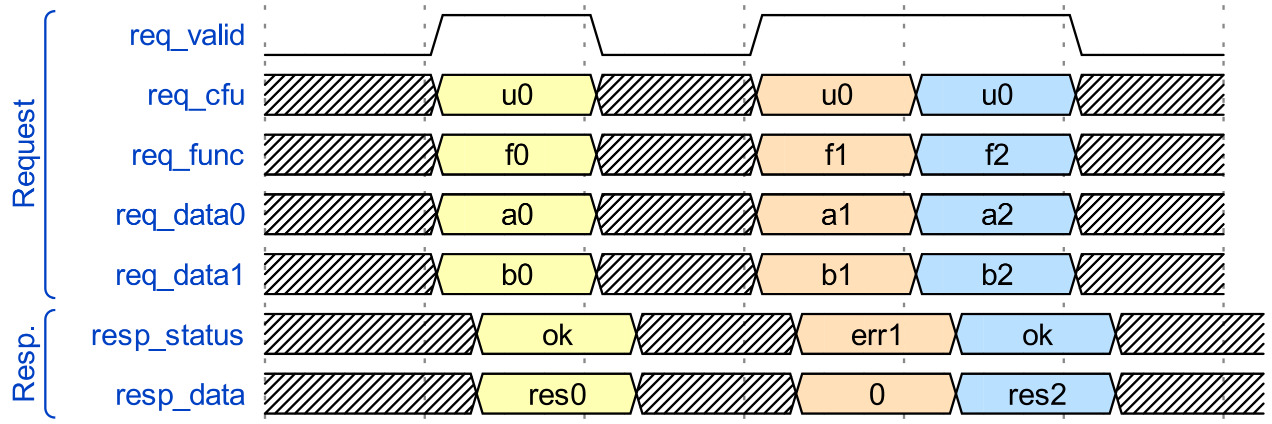
Level	Dir	Port	Width Parameter	Description
1+	in	`clk`		clock
1+	in	`rst`		reset
1+	in	`clk_en`		clock enable
	in	`req_valid`		request valid
2+	out	`req_ready`		request ready
3	in	`req_id`	`CXU_REQ_ID_W`	request REQ_ID
	in	`req_cxu`	`CXU_CXU_ID_W`	request CXU_ID
1+	in	`req_state`	`CXU_STATE_ID_W`	request STATE_ID
	in	`req_func`	`CXU_FUNC_ID_W`	request CF_ID
2+	in	`req_insn`	`CXU_INSN_W`	request raw instruction
	in	`req_data0`	`CXU_DATA_W`	request operand data 0
	in	`req_data1`	`CXU_DATA_W`	request operand data 1
1+	out	`resp_valid`		response valid
2+	in	`resp_ready`		response ready
3	out	`resp_id`	`CXU_REQ_ID_W`	response ID
	out	`resp_status`	`CXU_STATUS_W`	response status
	out	`resp_data`	`CXU_DATA_W`	response data

CXU-L0: combinational CXU *(§3.5)*

- Stateless composable extensions only
- Combinational function of the CXU request
- Send response after comb. delay



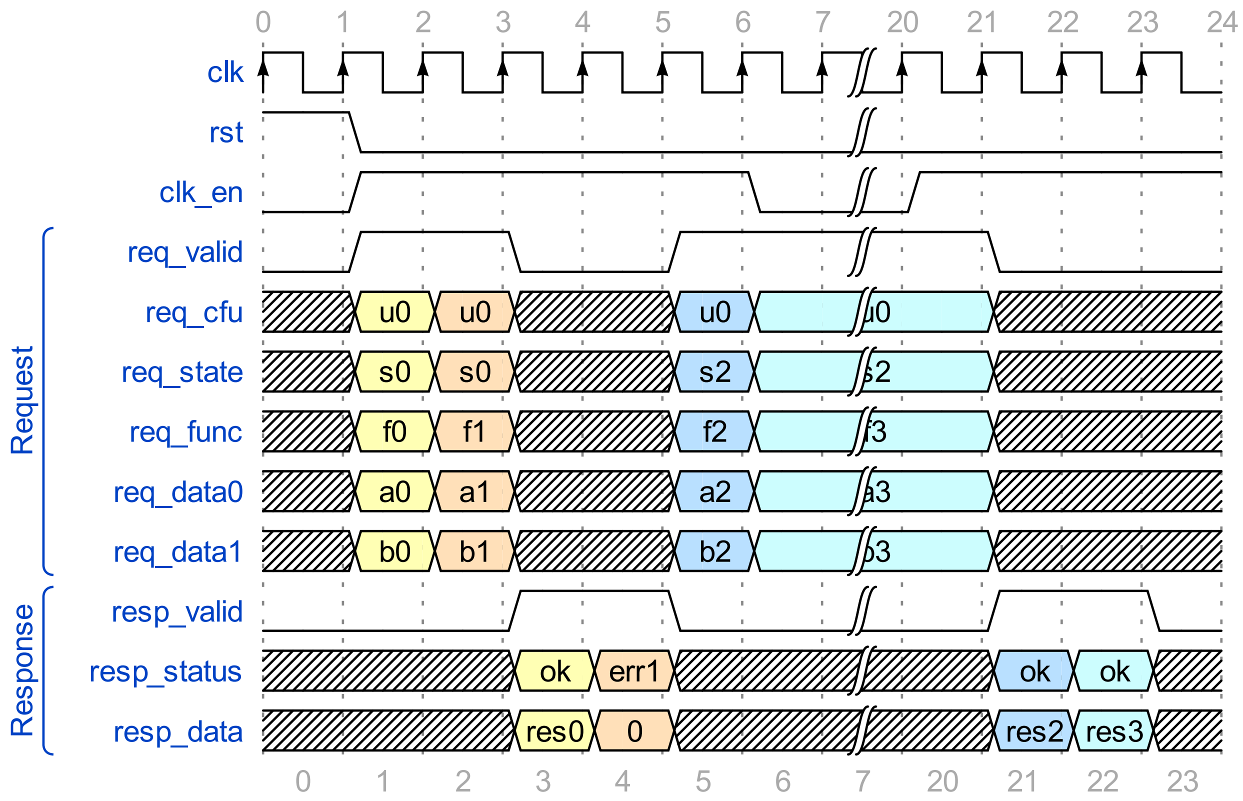
Dir	Port	Width Parameter	Description
in	req_valid		request valid
in	req_cxu	CXU_CXU_ID_W	request CXU_ID: selects
in	req_func	CXU_FUNC_ID_W	request CF_ID
in	req_data0	CXU_DATA_W	request operand data 0
in	req_data1	CXU_DATA_W	request operand data 1
out	resp_status	CXU_STATUS_W	response status
out	resp_data	CXU_DATA_W	response data



CXU-L1: fixed-latency (pipelined) CXU *(§3.6)*

- Compute function of request *and state*; send response after CXU_LATENCY cycles
- Initiation interval=1/cycle. No flow control
- Clock gating to save power: clk_en=0 suspends CXU interface and internals

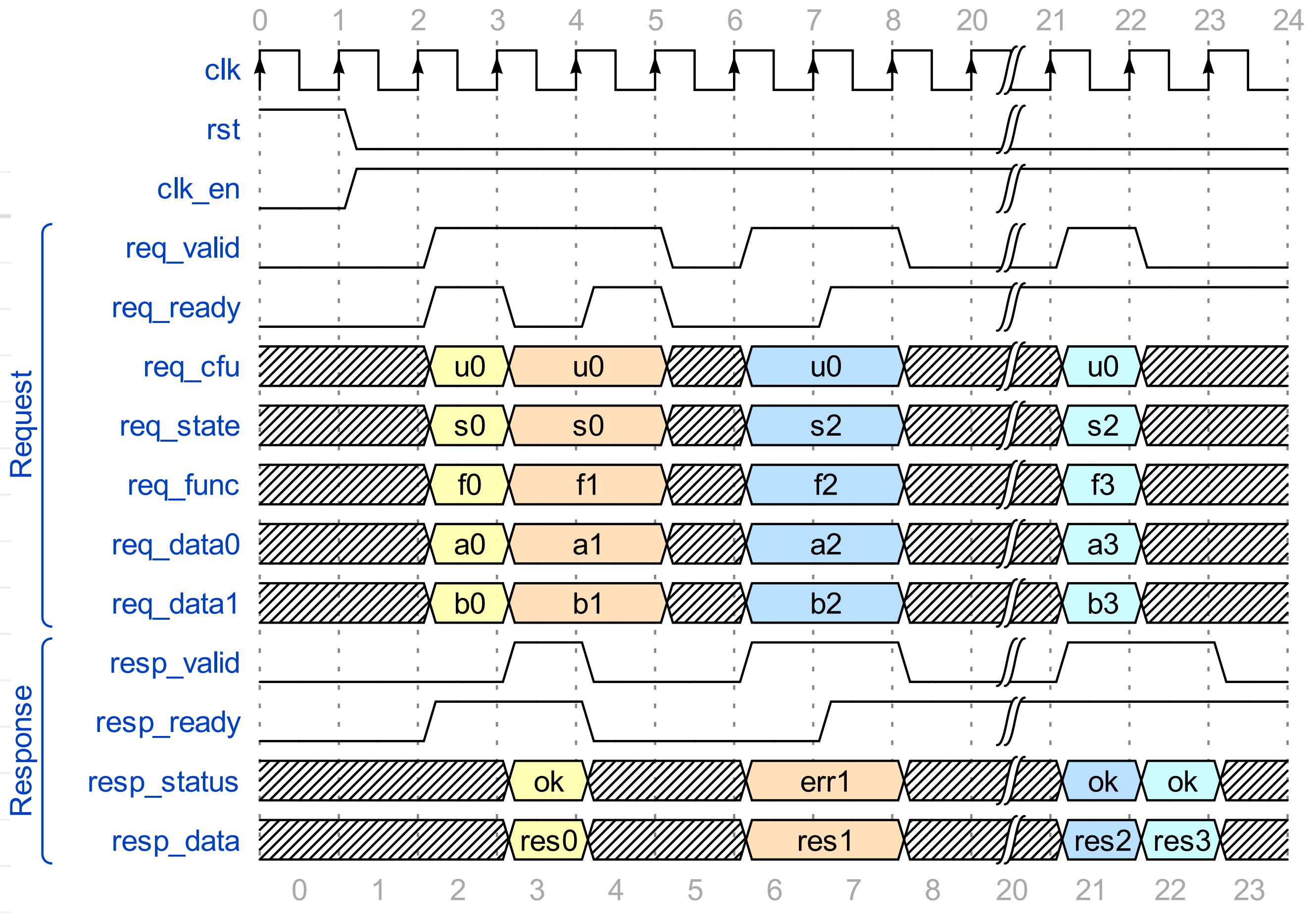
Dir	Port	Width Parameter	Description
in	`clk`		clock
in	`rst`		reset
in	`clk_en`		clock enable
in	`req_valid`		request valid
in	`req_cxu`	`CXU_CXU_ID_W`	request CXU_ID
in	`req_state`	`CXU_STATE_ID_W`	request STATE_ID
in	`req_func`	`CXU_FUNC_ID_W`	request CF_ID
in	`req_data0`	`CXU_DATA_W`	request operand data 0
in	`req_data1`	`CXU_DATA_W`	request operand data 1
out	`resp_valid`		response valid
out	`resp_status`	`CXU_STATUS_W`	response status
out	`resp_data`	`CXU_DATA_W`	response data



CXU-L2: variable-latency (stream) CXU *(§3.7)*

- Compute function of request and state; send response, in order, later
- Flow control: CXU can suspend requests; CPU can suspend responses

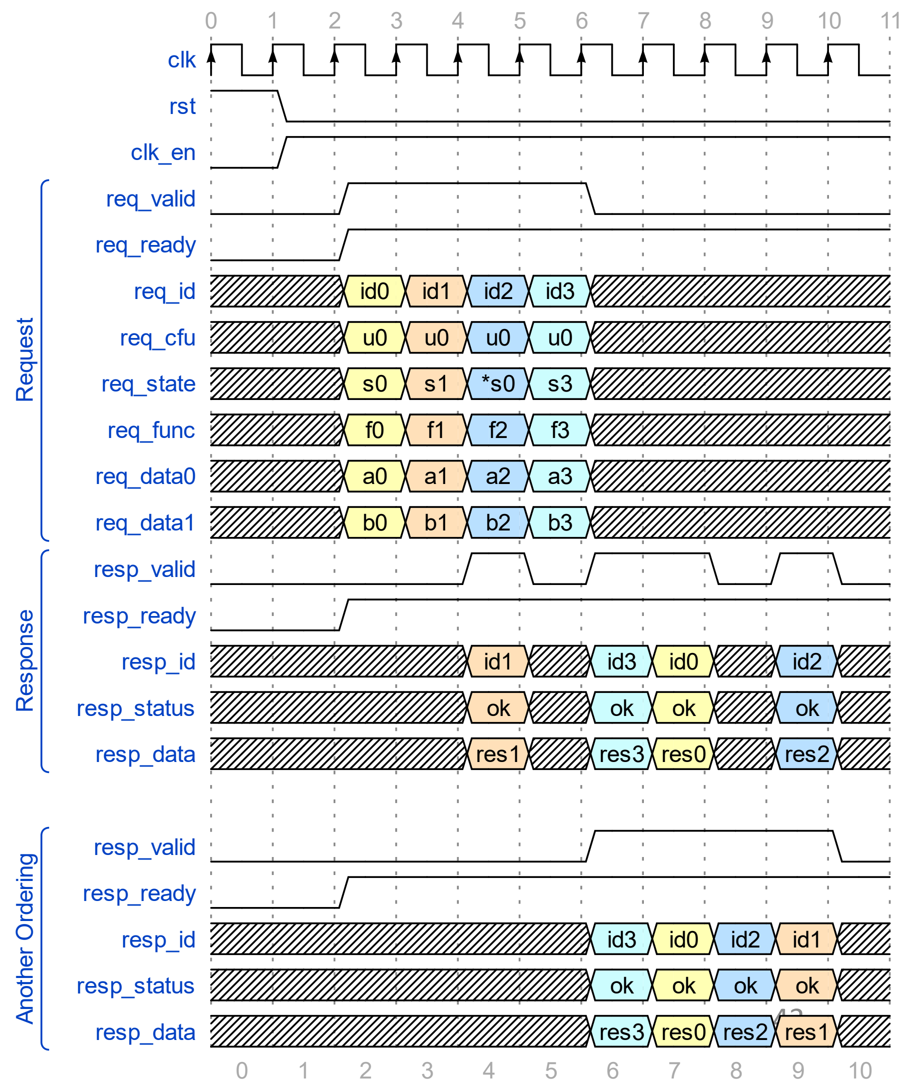
Dir	Port	Width Parameter	Description
in	clk		clock
in	rst		reset
in	clk_en		clock enable
in	req_valid		request valid
out	req_ready		request ready
in	req_cxu	CXU_CXU_ID_W	request CXU_ID
in	req_state	CXU_STATE_ID_W	request STATE_ID
in	req_func	CXU_FUNC_ID_W	request CF_ID
in	req_insn	CXU_INSN_W	request raw instruction
in	req_data0	CXU_DATA_W	request operand data 0
in	req_data1	CXU_DATA_W	request operand data 1
out	resp_valid		response valid
in	resp_ready		response ready
out	resp_status	CXU_STATUS_W	response status
out	resp_data	CXU_DATA_W	response data



CXU-L3: reordering CXU *(§3.8)*

- Compute func of request and state;
  send response, possibly out of order

Dir	Port	Width Parameter	Description
in	clk		clock
in	rst		reset
in	clk_en		clock enable
in	req_valid		request valid
out	req_ready		request ready
in	req_id	CXU_REQ_ID_W	request REQ_ID
in	req_cxu	CXU_CXU_ID_W	request CXU_ID
in	req_state	CXU_STATE_ID_W	request STATE_ID
in	req_func	CXU_FUNC_ID_W	request CF_ID
in	req_insn	CXU_INSN_W	request raw instruction
in	req_data0	CXU_DATA_W	request operand data 0
in	req_data1	CXU_DATA_W	request operand data 1
out	resp_valid		response valid
in	resp_ready		response ready
out	resp_id	CXU_REQ_ID_W	response ID
out	resp_status	CXU_STATUS_W	response status
out	resp_data	CXU_DATA_W	response data



CXU-LI-compliant CPUs *(§3.10)*

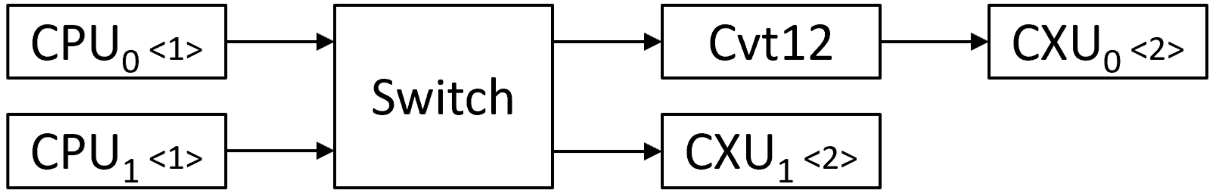
- -Zicx CX multiplexing: ⇨CXU requests, ⇦CXU reponses
- Composition: CXUs’ levels ≤ CPUs’ levels
- Austere single cycle CPU
  - -L0 combinational CXUs only
- Pipelined in-order CPU
  - -L1 for some fixed latency
  - -L2, suspend pipeline on req_ready=0, negate resp_ready when regfile WB busy
- OoO completion CPU
  - -L2 or -L3

CXU Feature Level Adapters *(§3.9)*

- Translate higher level requests/responses for lower level CXUs
- Configurable
- Cvt01: combinational -L0 ⇨ pipelined -L1
  - With CXU_LATENCY-configurable response { valid, status, data } shift register
- Cvt02: combinational -L0 ⇨ streaming -L2
  - With response register + valid-ready logic + suspend reqs while resps suspended
- Cvt12: pipelined -L1 ⇨ streaming -L2
  - With pending responses queue + suspend reqs while queue full & resps suspended
- See repo ‘zoo’

CXU Switch *(§3.11)*

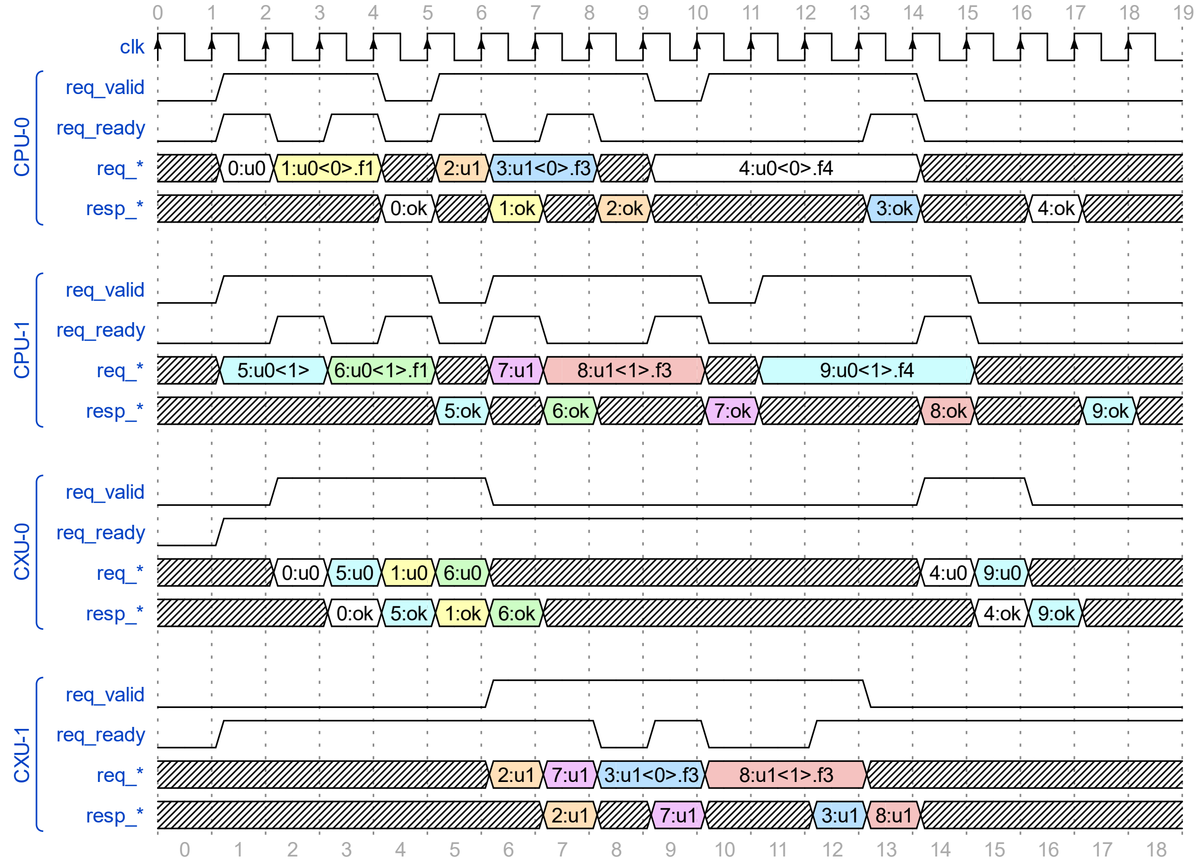
- Configurably compose 1+ initiators with 1+ targets
- Transfer *eligible* inputs to *available* outputs
  - Route responses back to request initiators
  - Ensure each initiator receives in-order responses



```
csrw mcx_selector,x20    ; select CXU_ID=0 and STATE_ID=HART_ID
cx_reg 0,x3,x1,x2        ; u0.f0
cx_reg 1,x6,x5,x4        ; u0.f1

csrw mcx_selector,x21    ; select CXU_ID=1 and STATE_ID=HART_ID
cx_reg 2,x9,x7,x8        ; u1.f2
cx_reg 3,x12,x11,x10     ; u1.f3

csrw mcx_selector,x20    ; select CXU_ID=0 and STATE_ID=HART_ID
cx_reg 4,x15,x13,x14     ; u0.f4
```



CXU-L2/L3 ⇔ AXI4 Streams

(§3.12)

Dir	CXU-LI Port	Width	AXI-S Port
in	`clk`		`aclk`
in	`rst`		`aresetn` *(inverted)*
in	`clk_en`		-
in	`req_valid`		`reqs_tvalid`
out	`req_ready`		`reqs_tready`
in	`req_id`	`CXU_REQ_ID_W`	`reqs_tid` *or* `reqs_tdest`
in	`req_cxu`	`CXU_CXU_ID_W`	`reqs_tuser` *or* `reqs_tdest`
in	`req_state`	`CXU_STATE_ID_W`	`reqs_tuser`
in	`req_func`	`CXU_FUNC_ID_W`	`reqs_tuser`
in	`req_insn`	`CXU_INSN_W`	`reqs_tuser`
in	`req_data0`	`CXU_DATA_W`	`reqs_tdata`
in	`req_data1`	`CXU_DATA_W`	`reqs_tdata`
in	-		`reqs_tlast` *optional*
in	-	*	`reqs_tstrb` *optional*
in	-	*	`reqs_tkeep` *optional*
out	`resp_valid`		`resps_tvalid`
in	`resp_ready`		`resps_tready`
out	`resp_id`	`CXU_REQ_ID_W`	`resps_tid` *or* `resps_tdest`
out	`resp_status`	`CXU_STATUS_W`	`resps_tuser`
out	`resp_data`	`CXU_DATA_W`	`resps_tdata`
out	-		`resps_tlast` *optional*
out	-	*	`resps_tstrb` *optional*
out	-	*	`resps_tkeep` *optional*

Metadata *(§4)*

Easier system composition with metadata

Preliminary – awaits revision

- Each CPU & CXU has metadata: properties, features, CXU-LI parameters
- Composer: Σ MDs + manifest ⇨ composed system RTL, CXU Map

Schema:

```
cxu_name: string
cxu_li:
    feature_level: scalar
    state_id_max: scalar | list | 'range'
    req_id_w: scalar | list | 'range'
    cxu_id_w: scalar | list | 'range'
    state_id_w: scalar | list | 'range'
    insn_w: scalar | list | 'range'
    func_id_w: scalar | list | 'range'
    data_w: scalar | list
    latency: scalar | list | 'range'
    reset_latency: scalar | list | 'range'
    xyz_range: [min,max]
```

Example CXU MD:

```
cxu_name: bobs_bnn_cxu
cxu_li:
    feature_level: 1
    state_id_max: 1         # only supports 1 state context
    req_id_w:               # any req_id is fine
    cxu_id_w: 0             # no req_cxu
    state_id_w: 0           # no req_state_id
    insn_w: 0               # no req_insn
    func_id_w: range        # need >= 5-bit CF_IDs
    func_id_w_range: [5,10] # so [5,6,7,8,9,10] are OK
    data_w: 64              # XLEN=64-bit only
    latency: [2,3,4]        # configurable w/ 2-4 cycles of latency
    reset_latency: 1        # requires at least 1 cycle of reset latency
other:
    adder_tree: [0,1]       # non-standard config parameter
    element_w: [4,8,16,32]  # non-standard config parameter
```

CX Runtime *(§TBD)*

CX Runtime API

Preliminary – awaits revision

```
typedef int128_t cx_id_t;               // global: CX ID, a 128b GUID
typedef int32_t cxu_id_t;               // system: CXU index
typedef int32_t state_id_t;             // system: state index
typedef int32_t cx_sel_t;               // hart:   CX selector (value or index)

interface cx_runtime {
  // CX discovery
  static cxu_id_t get_cxu(cx_id_t);     // map CX to its CXU here, if any

  // state context management
  static state_id_t alloc_state(cxu_id_t);
  static void free_state(cxu_id_t, state_id_t);

  // CX+state selectors
  static cx_sel_t alloc_sel_cxu(cxu_id_t);
  static cx_sel_t alloc_sel_cxu_state(cxu_id_t, state_id_t);
  static void free_sel(cx_sel_t);

  // CX multiplexing
  static cx_sel_t set_cur_sel(cx_sel_t); // return prev selector
};
```

Status, next steps

Status

Draft

- Spec: first 45 pages
- CXU Zoo, CXU adapters
    - -L0: popcount, bnn
    - -L1: mulacc, dotprod, cvt01
    - -L2: cvt02, cvt12, mux, switch

V1 TODOs

- Discuss in RVI sig-soft-cpu, consensus
- RVI ISA and non-ISA Task Groups
- Overhaul metadata
- Design & build CX Runtime, Composer
- Explore V2, so V1 doesn't preclude it
- -Zicx + CXU-LI CPU cores
- CX libraries for CXU Zoo
- Import CFU Playground CFUs into CXU Zoo
- **Demo: 1 CPU + 2 CXUs + libraries**
- **Plug Fest: CPUs × CXUs × libraries**
- Verification?

Call to Action

We propose two new RISC-V International Task Groups:

- **CX-ISA TG**: -Zicx ISA/CSRs to enable composable extensions
- **CXU-LI TG**: (non-ISA) logic interface for CX hardware

We submit:

- *Draft Proposed RISC-V Composable Custom Extensions Specification*
  https://github.com/grayresearch/CX

Questions? Suggestions? Criticism?
Discuss as github repo issues, or in RVI sig-soft-cpu group