CCX:
Composable Custom eXtensions

2023 RISC-V Summit Europe (Barcelona)

https://www.youtube.com/watch?v=ugr3RqmWhWw&list=PL85jopFZCnbMlDT5WOagUjQnk1ZSgjuRa&index=10&ab_channel=RISC-VInternational

Guy Lemieux (UBC) and Jan Gray (Gray Research)

Other Contributors

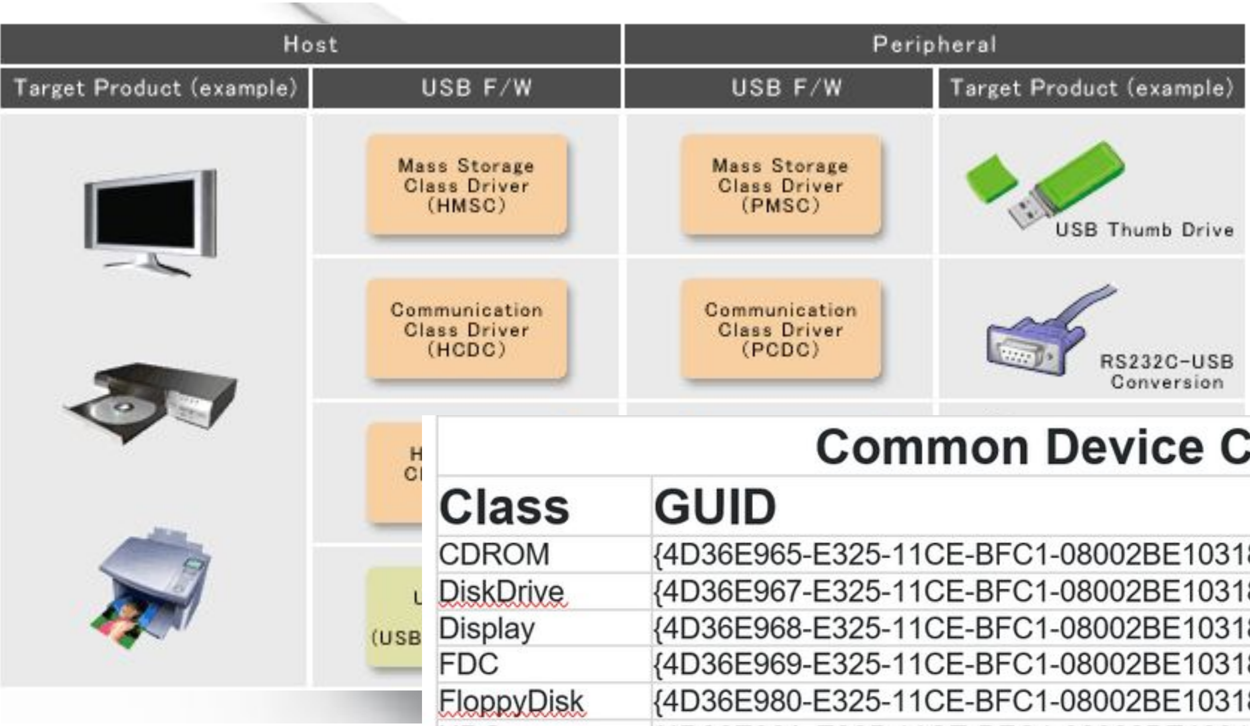
Tim Vogt (Lattice), Tim Callahan (Google), Charles Papon (SpinalHDL), Maciej Kurc (Antmicro), Karol Gugala (Antmicro), Olof Kindren (Qamcom)

The problem: incompatible interfaces

RISC-V
Processor

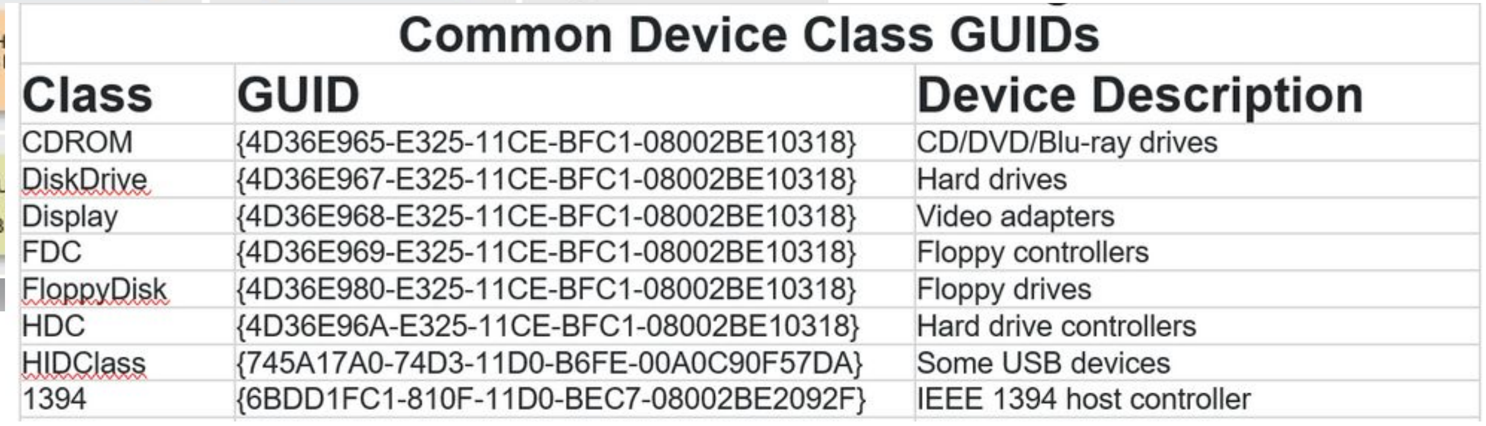
RISC-V
Accelerator

The problem (2): the stack



Host		Peripheral	
Target Product (example)	USB F/W	USB F/W	Target Product (example)
	Mass Storage Class Driver (HMSC)	Mass Storage Class Driver (PMSC)	USB Thumb Drive
	Communication Class Driver (HCDC)	Communication Class Driver (PCDC)	RS232C-USB Conversion
	H Cl		
	U (USB		

- Standard physical interface
- Needs standardized software stack



Common Device Class GUIDs		
Class	**GUID**	**Device Description**
CDROM	{4D36E965-E325-11CE-BFC1-08002BE10318}	CD/DVD/Blu-ray drives
DiskDrive	{4D36E967-E325-11CE-BFC1-08002BE10318}	Hard drives
Display	{4D36E968-E325-11CE-BFC1-08002BE10318}	Video adapters
FDC	{4D36E969-E325-11CE-BFC1-08002BE10318}	Floppy controllers
FloppyDisk	{4D36E980-E325-11CE-BFC1-08002BE10318}	Floppy drives
HDC	{4D36E96A-E325-11CE-BFC1-08002BE10318}	Hard drive controllers
HIDClass	{745A17A0-74D3-11D0-B6FE-00A0C90F57DA}	Some USB devices
1394	{6BDD1FC1-810F-11D0-BEC7-08002BE2092F}	IEEE 1394 host controller

The problem (3): scarce ISA encodings

- 32b opcodes → nearly out of space
  - Even RV32 **Standard Extension** encodings **will overlap**

    “the P and V extensions have overlapping functionality,
    yet are largely suited to different application domains,
    we suggest declaring them mutually incompatible.
    This avails each extension of the other's major opcode” *(ARC)*

- Problem is worse for accelerators

  → unlike Standard Extensions, custom instructions are unmanaged

Many accelerator interfaces

- RoCC (UC Berkeley Rocket)
- Core-V-XIF (OpenHW Group)
- SCAIE-V (TU Darmstadt)
- ACE (Andes)
- VCIX (SiFive for vectors)

…more…

Common ground?

These interfaces solve the same problem:

Define new custom ISA encodings

… to connect a few instruction-based Accelerators

… to a few CPUs

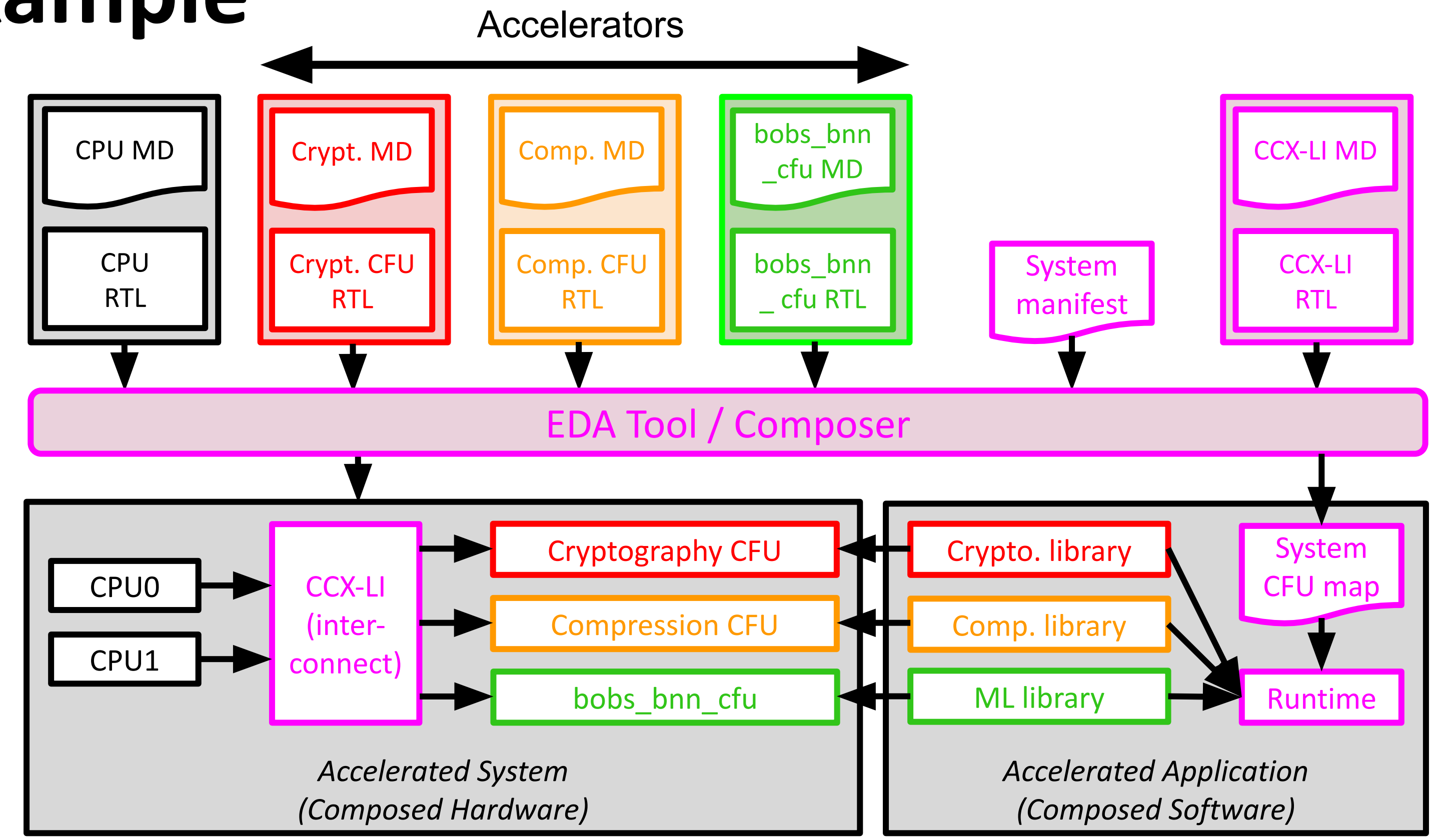
… to run software.

Why is this not enough?

Because your extensions won't compose with my extensions.

This continuously fragments the ecosystem!

Example



Accelerators
CPU MD
CPU RTL
Crypt. MD
Crypt. CFU RTL
Comp. MD
Comp. CFU RTL
bobs_bnn _cfu MD
bobs_bnn _ cfu RTL
System manifest
CCX-LI MD
CCX-LI RTL
EDA Tool / Composer
CPU0
CPU1
CCX-LI (inter-connect)
Cryptography CFU
Compression CFU
bobs_bnn_cfu
Crypto. library
Comp. library
ML library
System CFU map
Runtime
Accelerated System (Composed Hardware)
Accelerated Application (Composed Software)

A programming model

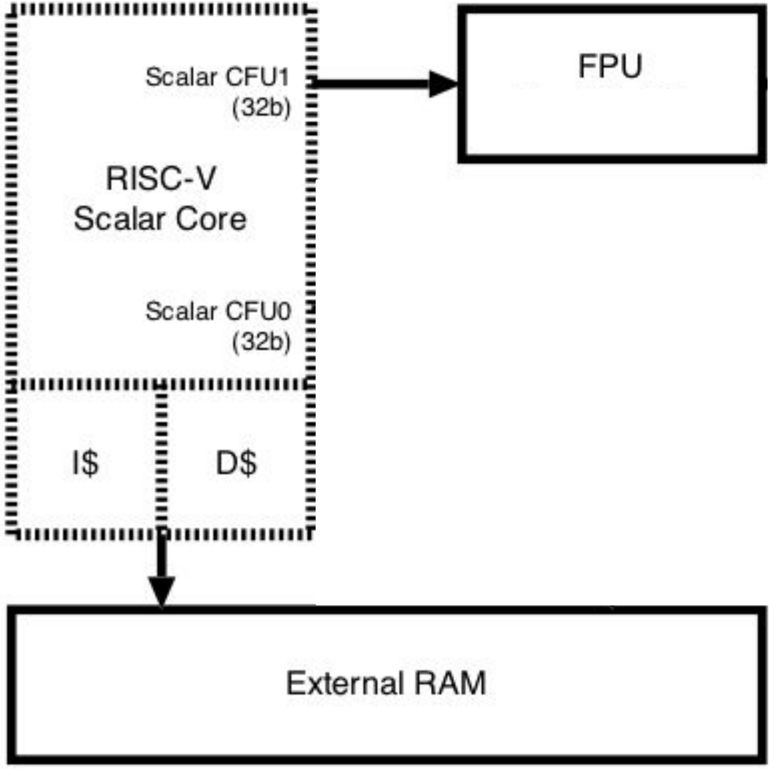
1. Discover
2. Select and allocate context (eg, constructor)
3. Issue custom instructions
4. Deselect (eg, destructor)

```
if (CI bm(GUID_BitmanipCI); bm)           // csrrw x2,mcfu_selector,x1
  count = custom0( POPCNT, data, 0 );     // custom-0 POPCNT,x3,x4,x5
  ~CI() deselection implied              // csrw mcfu_selector,x2
else
  count = popcount(data);                // unaccelerated software
```

Features

- Composability
  - Resolve ISA encoding conflicts
  - No centralized management
- Versioning & run-time discovery
- Multiple state contexts (& OS protected)
- Hierarchical nesting (figure on next slide)

Hierarchical accelerator composition

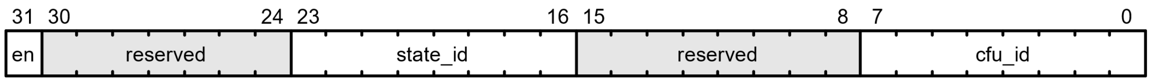


Suggestions: (1) consider merging cfu_reg and cfu_imm into custom-0, (2) consider “official” way of how to add official RISC-V extensions (eg, RVV) as a CCX implementation.

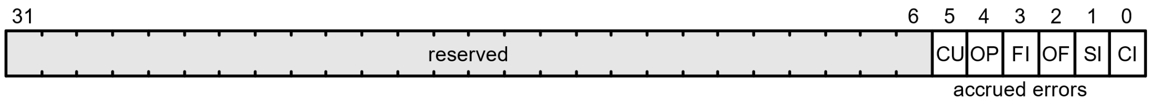
What’s needed?

- GUID (128b) – self-minted for each version of accelerator
- 2 CSRs
  - “Selector” to choose CFU_ID[7:0], STATE_ID[7:0]
  - “Status” to accrue errors

`mcfu_selector CSR`



`cfu_status CSR`



- Standardized instruction encodings

```
(custom-0 block)   cfu_reg   cf_id[10],rd,rs1,rs2
(custom-1 block)   cfu_imm   cf_id[7],rd,rs1,imm8
(custom-2 block)   cfu_flex cf_id[25]        // also sends [rs1], [rs2]
```

A programming model

1. Discover
2. Select and allocate context (eg, constructor)
3. Issue custom instructions
4. Deselect (eg, destructor)

```
if (CI bm(GUID_BitmanipCI); bm)              // csrrw x2,mcfu_selector,x1
  count = custom0( POPCNT, data, 0 );        // custom-0 POPCNT,x3,x4,x5
  ~CI() deselection implied                  // csrw mcfu_selector,x2
else
  count = popcount(data);                    // unaccelerated software
```

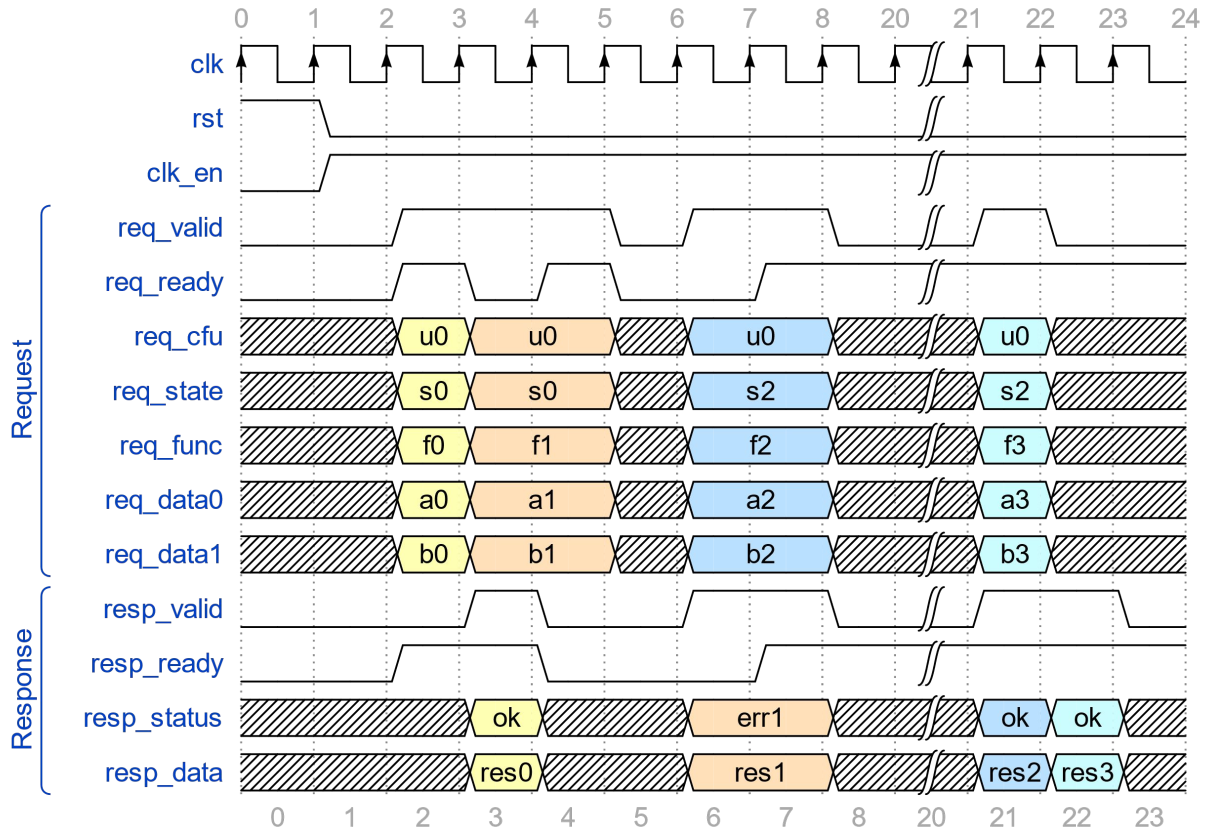
Privileged mode

Privileged mode needs 2 additional CSRs

- Protected CSR
  - CFU Selector Table
  - pointer to 4KiB page in DRAM (1024 entries)
  - each entry has < CFU index, STATE index >
- User CSR
  - offset to one of 1024 entries in CFU Selector Table

Physical Logic Interface

- Compatible with AXI-Stream



What else?

- Software
  - four custom-0 instructions to manage state
    - `cf_read_status`, `cf_read_state`
    - `cf_write_status`, `cf_write_state`
  - run-time system for discovery, allocation, context switching

- Chip building
  - **System manifest** describes CPU + accelerator connections
  - **Composer (EDA tool)** to connect CPUs + accelerators

Status

To Do

- Implementation proofs
    - CFU runtime
    - Composition tool
    - RTL/FPGA proofs (plugfest)
- Wishlist
    - Speculative (issue+squash)
    - Virtual memory
    - Coherence + consistency
    - Hypervisors

Call to Action

We propose two new Task Groups:

CCX-ISA: ISA and CSRs to enable <u>composable</u> custom extensions

CCX-LI: logic interface to physically connect accelerators with CPUs

To bootstrap this, we respectfully submit:

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