A Simple Plan: An API and ABI for Managing Multiple Distinct Sets of Custom Extensions

Brandon Freiberger[1], Jan Gray[2] and Guy G.F. Lemieux[1*]

[1]Dept of ECE, University of British Columbia
[2]Gray Research LLC

Abstract

The RISC-V custom instruction encoding space provides vendors a rich opportunity to innovate. Now, the pending Composable Extensions (CX) Task Group is planning to develop ISA and non-ISA specifications to avoid collisions in opcodes and provide uniform naming, discovery, error handling, context management, and other features that will enable vendors to create a marketplace for composition and reuse of their independently authored custom (instruction) extensions and their software libraries. Instruction set switching was presented in Barcelona [1] as a way to manage this problem. A critical missing piece of the puzzle is the end user perspective: what is the application API, how will the OS manage requests, and what are the implications on the ABI? This paper outlines a simple plan to address these issues, including allocating and virtualizing state context in Linux. ***We seek feedback from OS, compiler, applications and security perspectives.***

Switching Instruction Sets

The Draft CX Specification [2] describes a mechanism to multiplex the custom opcode space and custom CSR space. A set of custom instructions (and custom CSRs) that conforms to the specification is called a CX Extension (CXE). A given CXE may also have state (e.g., register file) as a single context or multiple contexts (copies). The specification has further details, including limitations on opcode encodings and functionality of custom instructions and custom CSRs.

The main problem is providing true binary software portability. Software needs to discover if a required CXE exists, and use its opcodes and CSRs in a conflict-free way, on any system (that may or may not implement this CXE or other CXEs). The solution should work on bare metal or an OS, and not rely upon recompilation, JIT, relinking, or binary rewriting. Use of an OS may provide enhanced functionality.

A central concept is the `CX_ID`. Used in the discovery and naming process, it is a 128b globally unique identifier (GUID) randomly assigned when a CXE is 'published'. After that, it describes an immutable set of custom opcodes, custom CSRs, and associated semantics; if this interface contract is ever changed, such as the addition of a new custom opcode, a new `CX_ID` is required. Collisions among random 128b keys are rare, so centralized management is not required.

To use a CXE, software must call a runtime library or OS to discover if its `CX_ID` is available in the system. If so, a *selector* is returned that uniquely points to that `CX_ID` and an allocated context. If not, software may provide its own emulation. This selector is used to activate the requested CXE and its context.

If all contexts are allocated, the runtime/OS may *virtualize* a context by spilling another physical context to memory. This is done by trapping on the first custom instruction after the active selector is changed.

In a managed OS, the selector is simply an index into a per-process 4KiB structure, called the `scx_table`, which contains up to 1023 raw selectors. The raw selectors are stored in OS-privileged memory page, thereby enforcing isolation of state between processes.

A Simple API

The proposed API follows a simple model: the user opens a context, selects one active context at a time, issues custom instructions to the active context, and closes a context when done. A process can have multiple open contexts, and can use them among all threads in an address space (cf. file descriptors).

The number of physical contexts is limited, perhaps even singular. However, the runtime/OS can save/restore this state, using an eager or lazy policy, thereby providing a large number of virtual contexts. The API provides users with control over context allocation, while leaving access control, virtualization and context save/restore to the runtime/OS.

There are 4 main functions in CX API:

- `cx_sel_t cx_open(cx_guid_t, cx_virtual_t, cx_sel_t);`
- `void     cx_select(cx_sel_t);`
- `void     cx_close(cx_sel_t);`
- `uint32_t cx_status(void);`

CX Open

This system call dynamically discovers an extension by `CX_ID` and allocates a context, if available; otherwise, it returns an error. This involves:

*Corresponding author: `lemieux@ece.ubc.ca`

- allocate a context
- allocate memory to spill the context in the kernel
- update the process' `scx_table`
- return the index to the table

The `cx_open` call enables virtualization of a physical context using the `cx_virtual_t` field. Any selector that has a virtualized (i.e., not exclusive) physical context is configured to generate an exception when the first custom instruction is executed after changing the active selector. This is done on lines 27, 30 and 33 in Figure 1. The trap handler saves the current physical context as the virtual context of the previous selector (which is known), and restores the virtual context of the current selector. Four virtualization priority levels give performance hints:

- **Priority 0: No Virtualization**. Returns a dedicated physical context or allocation fails.
- **Priority 1: Intra-Process Virtualization**. A process can share multiple virtual contexts to one physical context. Other processes cannot use the same physical context, limiting performance perturbations. An exclusive physical context is allocated when the previous selector parameter is -1. If none are available, it virtualizes any priority 1 context of the same process. If the previous selector parameter has a valid priority 1 context, it is virtualized, otherwise allocation fails.
- **Priority 2: Inter-Process Virtualization**. A process uses this to virtualize a physical context with other processes. There is no intra-process virtualization, so the previous selector parameter is ignored. An exclusive physical context is allocated, otherwise a context is virtualized with another priority 2 context from another process, otherwise allocation fails.
- **Priority 3: Full Virtualization**. A process uses this for highest likelyhood of a successful allocation by virtualizing with any priority 3 physical context. An exclusive physical context is allocated when the previous selector parameter is -1. If none are available, it will virtualize any other priority 3 context. If a valid priority 3 selector is provided, its physical context is virtualized, otherwise allocation fails.

CX Select

This fast call updates the currently active selector. It can be inlined, as it nominally consists of a single non-privileged instruction.

CX Close

This system call frees the underlying virtual or physical context and removes the selector from `scx_table`.

CX Status

This (usually) fast call does these things:

1. Completes all pending CX instructions (a fence)
2. Atomically reads and clears all bits in the CSR

This call can be inlined, as it nominally consists of a few non-privileged instructions. Details about the bits are described in the Draft CX Specification [2].

ABI Changes

An ABI that switches between two distinct instruction sets, namely 32b ARM and 16b Thumb, has been done since 1999 [3]. However, CX is more general, where only part of the ISA is switched among multiple CXEs, each CXE is unknown in advance, and each CXE may have its own physical context memory.

ABI changes must be minimized and maintain compatibility with legacy custom instructions, old binaries, and old libraries. New executables and libraries that use CXEs may change the active selector. The main issue is to keep the active selector correct at all times; a secondary issue is full backwards compatibility, particularly with legacy custom instructions. To achieve this, we use a caller-saved policy that conservatively falls back to selecting legacy custom instructions.

Our six ABI rules are presently manually enforced:

1. **ABI-INIT:** Initially, the selection is legacy mode.
2. **ABI-ENTRY:** On entry to an ordinary function, or following any function call, the selection is legacy mode.
3. **ABI-CX-ENTRY:** On entry to a CX function, the selection is CX mode.
4. **ABI-SELECT-CX:** Code must select a CX prior to issuing that CX's custom operations or calling a CX function.
5. **ABI-DESELECT-CX:** Code that selects a CX must select legacy mode prior to calling an ordinary function, returning, or stack unwinding.
6. **ABI-SELECT-LEGACY:** Code should select legacy mode before issuing legacy custom instructions.

Status and Limitations

Using QEMU, we have a bare metal and Linux API working. The Linux exception handler does context switching, but kernel scheduler changes and bare metal context switching are not yet implemented; these will be completed by the Summit. The ABI is implemented manually; compiler work is needed. We want a fast mechanism to emulate missing custom instructions (eg, when a CXE is not present in hardware) because exceptions are too slow for anything but debugging.

```c
#include <cx.h>

// normally in "cxe_macc.h"
#define CXE_MACC_GUID \
    0xFEDCBA98765432100123456789ABCDEF
#define CXE_MACC_FUNC_ID 42
__CX__ inline int my_macc(int a, int b)
{ // ABI rule 3
    return cx_reg( CXE_MACC_FUNC_ID, a, b );
}

// normally in "my_macc_test.c"
void my_macc_test()
{
    cx_select( CX_LEGACY ); // ABI rule 2

    // open two independent state contexts
    cx_sel_t selA, selB;
    selA = cx_open( CXE_MACC_GUID, 1, -1 );
    selB = cx_open( CXE_MACC_GUID, 1, selA );

    if ( selA <= 0 || selB <= 0 ) {
        cx_select( CX_LEGACY ); // ABI rule 5
        return do_software_implementation();
    }

    cx_select( selA ); // ABI rule 4
    int result = my_macc( 3, 5 ); // 15

    cx_select( selB ); // ABI rule 4
    result = my_macc( 5, 5 ); // 25

    cx_select( selA ); // ABI rule 4
    result = my_macc( 3, 3 ); // 24

    uint32_t rc = cx_status(); // sync+check

    cx_close( selA );
    cx_close( selB );

    cx_select( CX_LEGACY ); // ABI rule 5
}
```

Figure 1: *Example usage of the CX API and ABI, alternating between 2 selectors that virtualize a single physical context. The cx_reg() call inserts inline assembly, specifying the function ID plus two operands in the integer register file, and writing a result back to the integer register file. More details are in section 2.3.1 of the Draft CX Specification [2].*

References

[1] G. Lemieux and Jan Gray. *From CCX to CIX: A Modest Proposal for (Custom) Composable Instruction eXtensions for RISC-V*. Barcelona: RISC-V Summit Europe, 2023.

[2] J. Gray, G. Lemieux, et al. *Draft Proposed RISC-V Composable Custom Extensions Specification*. Version 0.95.240403. URL: `https://github.com/grayresearch/CX/blob/main/spec/spec.pdf` (visited on 07/06/2024).

[3] ARM Limited. *ARM Developer Suite Developer Guide*. Version Release 1.2. Nov. 2001. URL: `https://developer.arm.com/documentation/dui0056/d?lang=en`.