



THE UNIVERSITY
OF BRITISH COLUMBIA

CIA: Custom Instruction-based Accelerators

Brandon Freiberger

Joseph Maheshe

Guy Lemieux

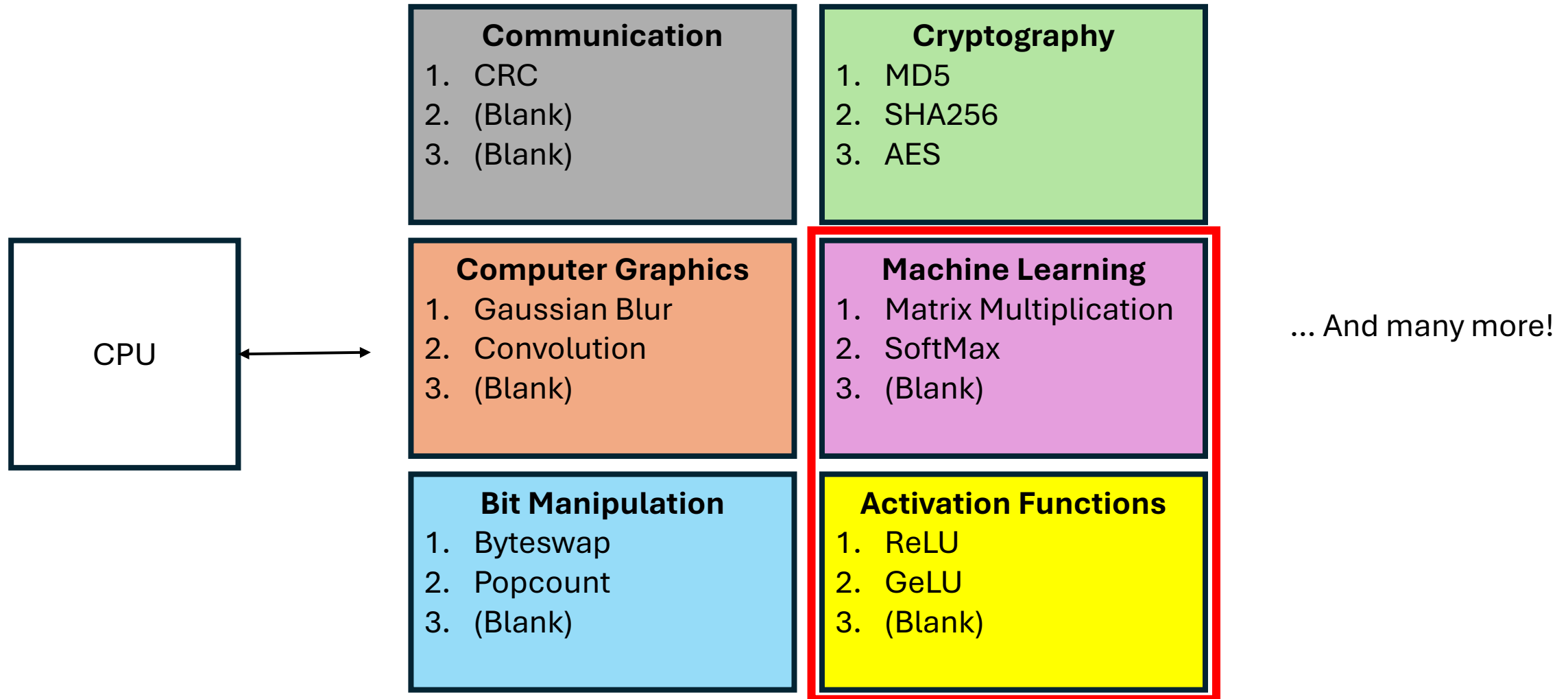
University of British Columbia, Vancouver

2026-09-03

Jan Gray

Gray Research LLC

Motivation

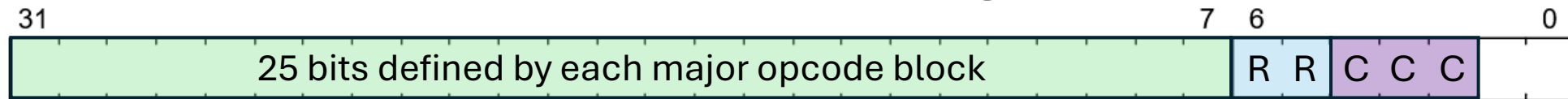


Background: RISC-V

- Instruction Set Architecture (ISA)
 - Instructions
 - Control status registers (CSRs)
- A core design principle of RISC-V is modularity
 - Allows for separation of concerns
- Defines dozens of standard extension
- Supports vendor-specific **custom extensions**

Background: RISC-V Major Opcodes

Instruction encoding:



	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	OP-VE	custom-3/rv128	≥80b

Background: Draft CX Specification

Definitions:

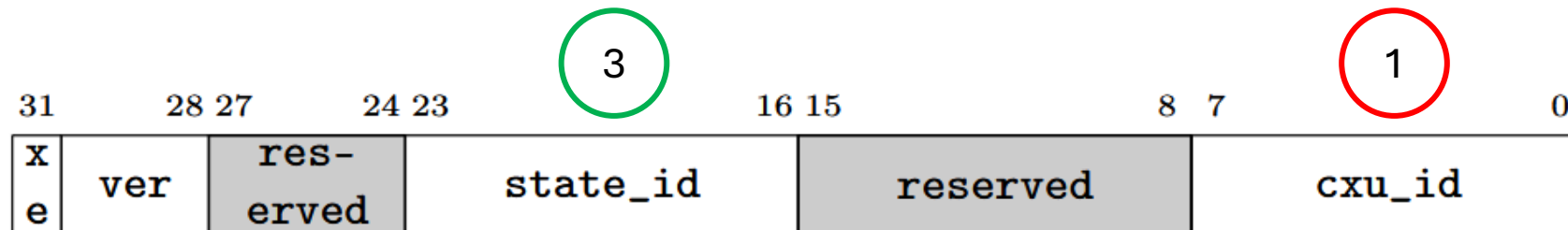
- A Composable Custom eXtension (CX)
 - **Immutable ISA contract**
 - Set of custom instructions + CSRs, their state, and their behavior
 - Implemented as a **Custom Extension Unit (CXU)**
- Each CX is identified by 128 bit UUID
 - CX authors can generate a random 128 bit UUID
 - Changes to CX ISA require new UUID
- A CPU system map between UUID and cxu_id

CPU Core	cxu_id = 0	UUID = 0xAA	Machine Learning
	cxu_id = 1	UUID = 0xBB	Activation Functions

Background: Draft CX Specification

The Draft CX Specification defines:

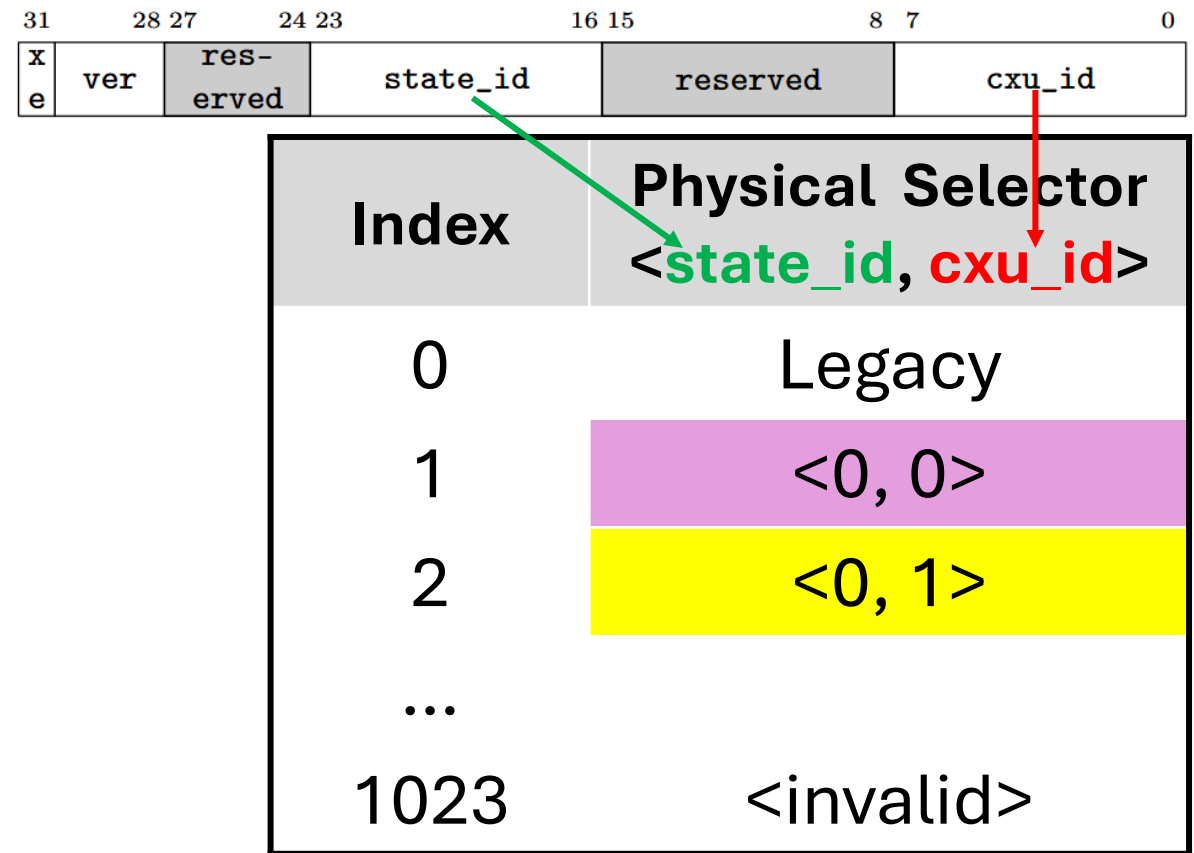
1. Bank switching of **CXUs** (accelerators)
2. CXU Status
3. Bank switching of **state contexts**
4. Virtualization and Protection/Isolation (Table vs. Direct)
5. ABI (non-ISA)
6. Optional logic interface (non-ISA)



Physical Selector

Background: Selectors

- Table Mode (Draft CX Specification)
 - 4kB Memory Table (System-level)

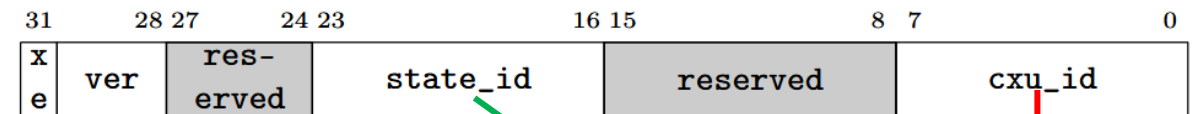


Background: Selectors

- Table Mode (Draft CX Specification)
 - 4kB Memory Table (System-level)
 - Opaque selector (User-level)



Opaque User-Level Selector



Index	Physical Selector <state_id, cxu_id>
0	Legacy
1	<0, 0>
2	<0, 1>
...	
1023	<invalid>

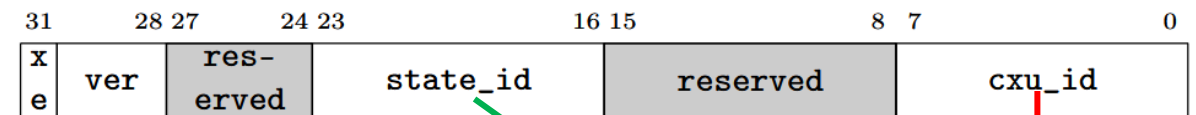
4kB Selector Table 8

Background: Selectors

- Table Mode (Draft CX Specification)
 - 4kB Memory Table (System-level)
 - Opaque selector (User-level)
- Provides OS-level protection/isolation
- Unforgable: Users can only access physical selectors in the Table created by the OS



Opaque User-Level Selector



Index	Physical Selector <state_id, cxu_id>
0	Legacy
1	<0, 0>
2	<0, 1>
...	
1023	<invalid>

4kB Selector Table

What's Missing?

- **No programming model**

What's Missing?

- **No programming model**
- **No model for managing state**

What's Missing?

- **No programming model**
- **No model for managing state**

Introducing Custom Instructions for Accelerators (CIA) API

API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}
```

API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}
```

API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}
```

API: Example Use Case

```
// main.c (user code)
```

```
#include "my_pow.h"
```

```
#include "my_macc.h"
```

```
int main() {
```

```
...
```

```
int x, sum;
```

```
for ( int i = 0; i < N; i++ ) {
```

```
    x = my_pow( i+2, 2 );
```

```
    sum = my_macc( x, x );
```

```
}
```

```
my_pow_close();
```

```
my_macc_close();
```

```
...
```

```
}
```

```
// my_pow.c (library source code)
```

```
#include "my_pow.h"
```

```
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */};
```

```
static cia_sel_t sel = -1; // Persistent Selector
```

```
int my_pow( int a, int b ) {
```

```
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
```

```
    if ( sel < 0 ) { /* return do_software()... */ };
```

```
    cia_select( sel ); // Inline CSRW
```

```
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic
```

```
    cia_select( CX_LEGACY );
```

```
    return val;
```

```
}
```

```
void my_pow_close() { cia_close( sel ); sel = -1; }
```

API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}

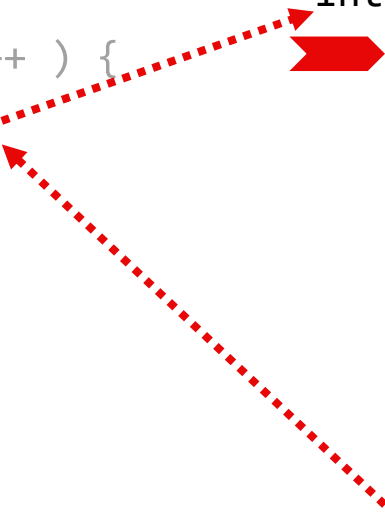
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // Persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}

void my_pow_close() { cia_close( sel ); sel = -1; }
```



API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}

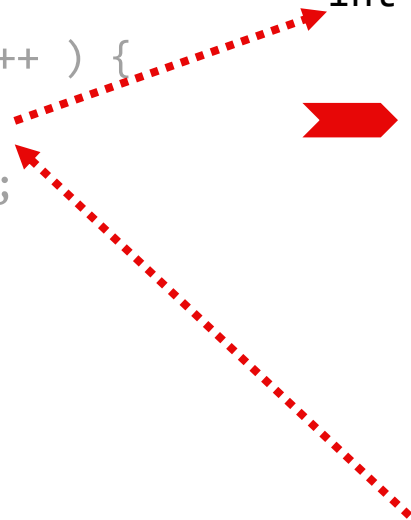
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}

void my_pow_close() { cia_close( sel ); sel = -1; }
```



API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}

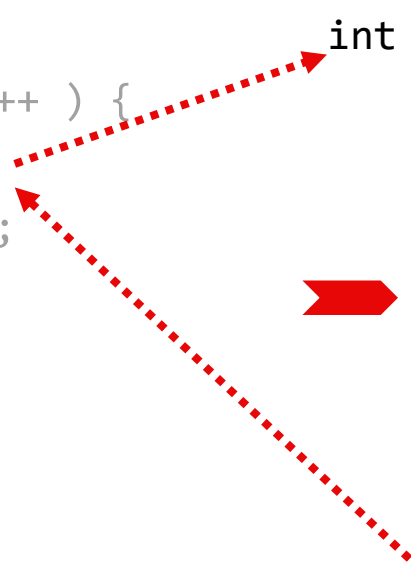
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // Persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}

void my_pow_close() { cia_close( sel ); sel = -1; }
```



API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}

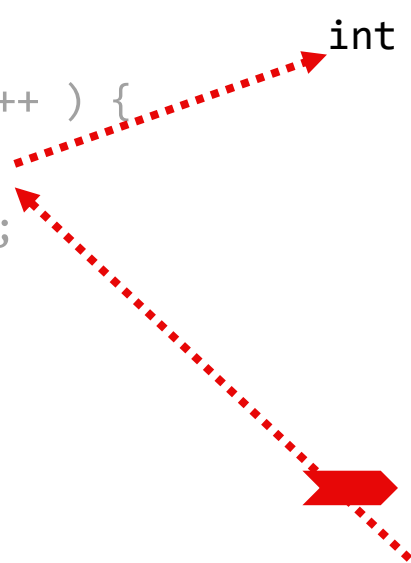
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // Persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}

void my_pow_close() { cia_close( sel ); sel = -1; }
```



API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}
```

```
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // Persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}

void my_pow_close() { cia_close( sel ); sel = -1; }
```

API: Example Use Case

```
// main.c (user code)
#include "my_pow.h"
#include "my_macc.h"
int main() {
    ...
    int x, sum;
    for ( int i = 0; i < N; i++ ) {
        x = my_pow( i+2, 2 );
        sum = my_macc( x, x );
    }
    my_pow_close();
    my_macc_close();
    ...
}
```

```
// my_pow.c (library source code)
#include "my_pow.h"
const cx_uuid_t MY_POW_CX_UUID = { /* 128b value */ };
static cia_sel_t sel = -1; // Persistent Selector

int my_pow( int a, int b ) {
    if ( sel < 0 ) sel = cia_open( MY_POW_CX_UUID, ... );
    if ( sel < 0 ) { /* return do_software()... */ };

    cia_select( sel ); // Inline CSRW
    int val = cx_reg( OP_POW, a, b ); // Inline R-Type Intrinsic

    cia_select( CX_LEGACY );
    return val;
}
```

```
void my_pow_close() { cia_close( sel ); sel = -1; }
```

API Summary

```
cia_sel_t cia_open( cx_uuid_t, ... );
```

- Dynamic discovery
- Permission checking
- Allocation
- Initialization
- Selector generation

```
void cia_select( cia_sel_t );
```

- Activates a selector

```
void cia_close( cia_sel_t );
```

- Frees any resources allocated with `cia_open()`;

```
cia_error_t cia_status( void );
```

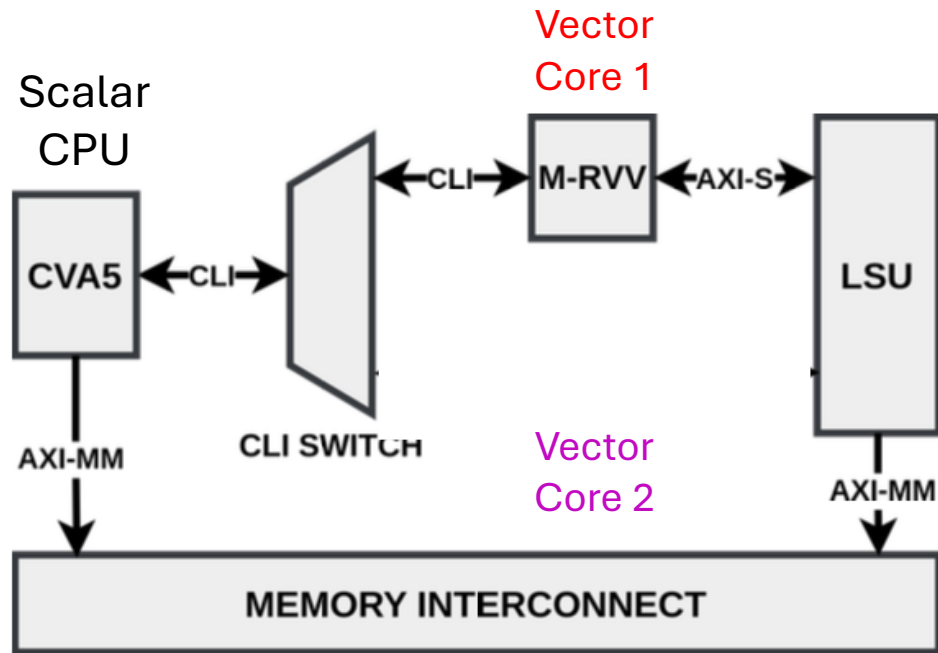
Sample Application: RGBA to LUMA Conversion

```
#include <cia.h> // CIA API
const int N=2;      // number of vector units
const int P=2;      // physical state contexts per unit
const int T=(N*P);  // total number of contexts
const int IMG_H=128; // image height
const int IMG_W=128; // image width
void rgba2luma( uint8_t *luma, uint32_t *rgba,
                cia_sel_t *selectors, int num_mthreads ) {
    assert( IMG_W <= 4*RV_VLEN/32 ); // simplifying assumption
    for( int i=0; i < IMG_H/num_mthreads; ++i ) {
        for( int t=0; t<num_mthreads; ++t ) {
            cia_select( selectors[t] ); // distributes rows
            // hidden: 17 vector instructions to process a row
            rgba += IMG_W;
            luma += IMG_W;
        }
        cia_select( CIA_TRADITIONAL );
    }
}
int benchmark( uint8_t *luma, uint32_t *rgba ) {
    cia_sel_t selectors[T];
    for ( int i=0; i<T; ++i )
        selectors[i] = cia_open( MRVV_UUID, CIA_NO_VIRT, -1 );
    rgba2luma( luma, rgba, selectors, T );
    for ( int i=0; i<T; ++i )
        cia_close( selectors[i] );
    return 0;
}
```

Configuration	FPGA runtime in kCycles (speedup)		
	rgba2luma	jacobi	median
Single thread, single vector core			
scalar (c)	469 (0.20×)	342 (0.21×)	5730 (0.04×)
scalar (w)	380 (0.24×)	222 (0.32×)	5690 (0.04×)
vector ideal	64 (1.45×)	66 (1.10×)	201 (1.20×)
vector prefetch	70 (1.32×)	71 (1.02×)	231 (1.04×)
CIA using t microthreads $\rightarrow n$ vector units $\times p$ physical contexts			
1 $\rightarrow$ 1 $\times$ 1	93 (1.00×)	72 (1.00×)	241 (1.00×)
2 $\rightarrow$ 1 $\times$ 2	70 (1.33×)	70 (1.03×)	208 (1.16×)
4 $\rightarrow$ 1 $\times$ 4	68 (1.36×)	67 (1.07×)	203 (1.19×)
8 $\rightarrow$ 1 $\times$ 8	71 (1.31×)	70 (1.03×)	203 (1.19×)
2 $\rightarrow$ 2 $\times$ 1	47 (1.98×)	37 (1.95×)	122 (1.98×)
4 $\rightarrow$ 2 $\times$ 2	35 (2.66×)	36 (2.00×)	105 (2.30×)
8 $\rightarrow$ 2 $\times$ 4	35 (2.66×)	35 (2.06×)	104 (2.32×)

Listing 3: Vectorized RGBA to LUMA conversion using CIA

Performance with 1 to 2 Vector cores



Multiple contexts hide memory latency

Vector Core 2 adds capacity

Configuration	FPGA runtime in kCycles (speedup)		
	rgba2luma	jacobi	median
Single thread, single vector core			
scalar (c)	469 (0.20×)	342 (0.21×)	5730 (0.04×)
scalar (w)	380 (0.24×)	222 (0.32×)	5690 (0.04×)
vector ideal	64 (1.45×)	66 (1.10×)	201 (1.20×)
vector prefetch	70 (1.32×)	71 (1.02×)	231 (1.04×)
CIA using t microthreads $\rightarrow n$ vector units $\times p$ physical contexts			
$1 \rightarrow 1 \times 1$	93 (1.00×)	72 (1.00×)	241 (1.00×)
$2 \rightarrow 1 \times 2$	70 (1.33×)	70 (1.03×)	208 (1.16×)
$4 \rightarrow 1 \times 4$	68 (1.36×)	67 (1.07×)	203 (1.19×)
$8 \rightarrow 1 \times 8$	71 (1.31×)	70 (1.03×)	203 (1.19×)
$2 \rightarrow 2 \times 1$	47 (1.98×)	37 (1.95×)	122 (1.98×)
$4 \rightarrow 2 \times 2$	35 (2.66×)	36 (2.00×)	105 (2.30×)
$8 \rightarrow 2 \times 4$	35 (2.66×)	35 (2.06×)	104 (2.32×)

Problem: Physical State Context Scarcity

- User program may need multiple state contexts of an accelerator
 - Example: crypto accelerator is used by main, library1, library2
 - main, library1, library2 are compiled separately (needs private state)

Problem: Physical State Context Scarcity

- User program may need multiple state contexts of an accelerator
 - Example: crypto accelerator is used by main, library1, library2
 - main, library1, library2 are compiled separately (needs private state)
- Problems
 - More libraries than physical contexts (intra-process)
 - More processes than physical contexts (inter-process)

Problem: Physical State Context Scarcity

- User program may need multiple state contexts of an accelerator
 - Example: crypto accelerator is used by main, library1, library2
 - main, library1, library2 are compiled separately (needs private state)
- Problems
 - More libraries than physical contexts (intra-process)
 - More processes than physical contexts (inter-process)
- We need to share (virtualize) state contexts
 - Time-multiplex multiple logical contexts into one physical context
 - Requires saving and restoring to memory
 - Context switching should be automated

Context Switching Challenges

- Goals
 - `cia_select()` call should be fast
 - Process switches should remain fast

Context Switching Challenges

- Goals
 - `cia_select()` call should be fast
 - Process switches should remain fast
- Two Naïve Solutions
 1. Intra-process: `cia_select()` saves and restores state with system call
 - Dramatically increases `cia_select()` cost
 2. Inter-process: Saving and restoring state on a process switch (eager)
 - Multiple CXUs, multiple contexts

State Context Switching Model

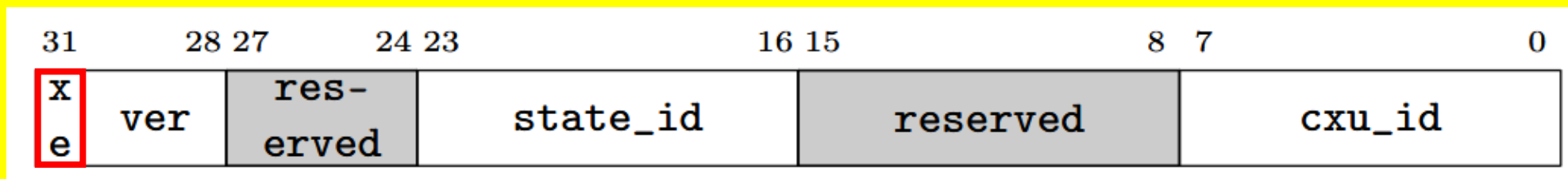
- Solution: delay context save + restore until necessary (lazy)
 - `cia_select()` only updates the selector

State Context Switching Model

- Solution: delay context save + restore until necessary (lazy)
 - `cia_select()` only updates the selector
- Saving and restoring is handled by the OS

State Context Switching Model

- Solution: delay context save + restore until necessary (lazy)
 - `cia_select()` only updates the selector
- Saving and restoring is handled by the OS
- Use exception enable bit to indicate first use
 - Works with both inter- and intra-process virtualization
 - Minimal added latency for process switches



CIA API Evaluation

API Instruction Count Overhead (QEMU)

	Linux QEMU
CIA API Call	Table
cia_open ^{1,2}	1373
cia_close	470
cia_sel	2
Custom Instr. Trap ²	683

¹ First use only; later calls may use more instructions.

² Varies with context size.

Results: Example Application

- Instructions per element for an example neural network (QEMU)
 - Using three CXs: **ML + Act** **ML** **Act**

```
for (i)
  for (j)
    for (k)
      C[i,j] += A[i,k] * B[k,j]
    C[i,j] = max(0, C[i,j]);
```



CIA Mode	Scalar
Table	5.95
Direct	5.95

Results: Example Application

- Instructions per element for an example neural network (QEMU)
 - Using three CXs: **ML + Act** **ML** **Act**

```
for (i)
  for (j)
    for (k)
      C[i,j] += A[i,k] * B[k,j]
    C[i,j] = max(0, C[i,j]);
```

```
cia_select(mulacc_act)
for (i)
  for (j)
    for (k)
      mac(A[i,k], B[k,j])
    C[i,j] = relu(read_acc())
```

CIA Mode	Scalar	Single CXU
Table	5.95	5.19
Direct	5.95	5.19

Results: Example Application

- Instructions per element for an example neural network (QEMU)
 - Using three CXs: **ML + Act** **ML** **Act**

```
for (i)
  for (j)
    for (k)
      C[i,j] += A[i,k] * B[k,j]
    C[i,j] = max(0, C[i,j]);
```

```
cia_select(mulacc_act)
for (i)
  for (j)
    for (k)
      mac(A[i,k], B[k,j])
      C[i,j] = relu(read_acc())
```

```
for (i)
  for (j)
    for (k)
      mac(A[i,k], B[k,j])
      C[i,j] = read_acc()
      cia_select(activation)
      C[i,j] = relu(C[i,j])
      cia_select(mulacc)
```

CIA Mode	Scalar	Single CXU	Two CXUs
Table	5.95	5.19	5.35
Direct	5.95	5.19	5.35

Takeaways

CIA API Provides a programming model and state model

Takeaways

CIA API Provides a programming model and state model

- **Ease of Use**

- Similar programming model to file descriptors
- Allows each library to have an independent, persistent selector
- Selectors automate virtualization and context switching

Takeaways

CIA API Provides a programming model and state model

- **Ease of Use**

- Similar programming model to file descriptors
- Allows each library to have an independent, persistent selector
- Selectors automate virtualization and context switching

- **Efficient**

- 2 instructions to switch between active extensions
- Lazy context save and restore

Limitations

- Current model only supports single core
- Unable test supervisor modes on hardware (currently)

Questions

Loop Instruction Count

Configuration	# inner loop instructions
Scalar	22
Single CXU	19
Two CXU	19

Configuration	# middle loop instructions
Scalar	14
Single CXU	13
Two CXU	17

Process Switch Overhead

Mode	Process Save	Process Restore
Direct	~10	~10
Table	~15	~15

Virtualization Types

`cia_sel_t`

```
cia_open( cx_uuid_t, cia_virt_t, cia_sel_t );
```

4 types of virtualization supported via `cia_virt_t`:

- No Virtualization (Best performance)
- Intra-process virtualization ()
- Inter-process virtualization ()
- Full Virtualization (Worst performance, fully shared)

Intra-Process Virtualization

1

```
// selA: 0, phys.context: 0
```

```
cia_sel_t selA = cia_open( MY_CX_UUID, CIA_INTRA_VIRT, -1 );
```

```
// selB: 1, phys.context: 1
```

```
cia_sel_t selB = cia_open( MY_CX_UUID, CIA_INTRA_VIRT, -1 );
```

```
// =====
```

2

```
// selA: 0, phys.context: 0
```

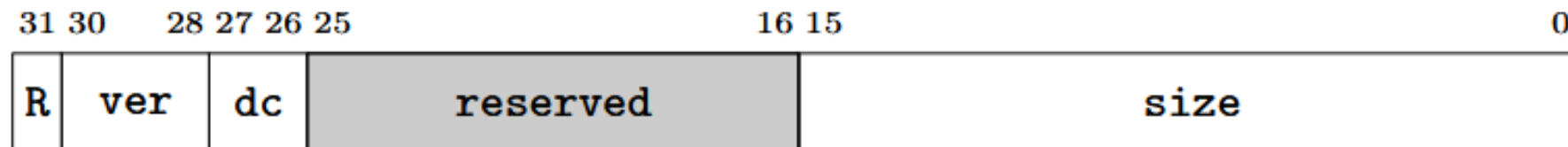
```
cia_sel_t selA = cia_open( MY_CX_UUID, CIA_INTRA_VIRT, -1 );
```

```
// selB: 1, phys.context: 0
```

```
cia_sel_t selB = cia_open( MY_CX_UUID, CIA_INTRA_VIRT, selA );
```

Context Save and Restore

- Each context has a `state_context_status_word` register
 - Used for context initialization + save/restore
- Draft CX Specification reserves instructions for read / write state
 - Issue a read / restore state instruction for each index



Bare Metal API

	Linux QEMU		FPGA
CIA API Call	Table	Direct	Bare Metal
<code>cia_open</code> ^{1,2}	1373	777	459
<code>cia_close</code>	470	277	105
<code>cia_sel</code>	2	2	38
Custom Instr. Trap ²	683	758	N/A

Bare Metal Instruction + Cycle counts

Configuration	Cycle Count	Insn. Count Per Element
Scalar	1,051,198	5.95
Single CXU	2,221,095	5.18
Separate CXUs	2,692,635	7.61

CIA API Block Diagram

