An Engines ISA Extension for the RISC-V Architecture

Eric L. McCorkle

Overview

The Engines ISA extension provides an instruction set for controlling hardware resources (either on- or off-die)

- Hardware resources can vary from implementation to implementation
- Virtualization and isolation features allow control without crossing privilege levels
- States/transitions control resource utilization profiles
- Interface allows for software-emulated engines as well

History

- Began as an attempt to design a crypto extension
- Design evolved into generic cipher “engines”
- Eventually became generic “engine” interface
- Discussions on RISC-V `isa-dev` list suggested modifications
- Extension evolved to add direct memory access capability
- Current iteration looks increasingly like an I/O instruction set

Hardware Additions

- *Engine Handle* register bank (32 $\times$ XLEN)
- Arbitrary number of *engine resources* (can be per-core or shared)
- *Ownership mask* and *access mask* CSRs, each with one bit per engine resource
- Engine resource descriptions (format TBD)

Engine Management

Instruction	Description
Function	
`acquire` *eh*,*type*	Acquire engine
Acquire an engine resource of type *type*, bind it to *eh*	
`release` *eh*	Release engine binding
Release the binding *eh*	
`ehsave` *rd*,*eh*	Save engine binding
Save the engine binding in *eh* to *rd*	
`ehbind` *eh*,*rs*	Load engine binding
Restore the engine handle *eh* from *rs*	

Engine management instructions allow engine resources to be acquired, released, and managed. Additionally, `ehbind` can be used to set up software-emulated engine handles, and to copy bindings to other engine handle registers.

States and Transitions

State	Description
`uninit`	Engine is uninitialized
`ready`	Engine is initialized, but not running
`running`	Engine is actively running
`paused`	Engine is paused, but can be quickly resumed
`saving`	Engine is being saved
`resuming`	Engine is being restored

Start States	Transition	End State
`uninit`	INIT	`ready`
`ready`, `saving`	RESET	`uninit`
`ready`	START	`running`
`running`	STOP	`ready`
any	PAUSE	`paused`
`paused`	UNPAUSE	state from previous pause
`paused`	SAVEBEGIN	`saving`
`saving`	SAVEEND	`paused`
`uninit`	RESBEGIN	`resuming`
`resuming`	RESEND	`paused`

States exist to handle initialization, resource consumption, and save/restore. Transitions may perform some amount of computation as part of their function.

Applications

There are a number of applications for the Engines instruction set:

- Device I/O instruction set
- Fast direct-access hardware interface in an OS
- Interface to on-die FPGAs
- Application-specific accelerators (essentially on-die ASICs) for many purposes
  - Cryptography (original purpose)
  - High-precision calculations
  - Expensive computations (scientific, financial, etc)
- Specialized coprocessor interfaces
- Foundation for other extensions

Future

- Revise greenfield instruction encoding, devise brownfield encoding
- Update GNU assembler/disassembler modifications
- Modify Rocket pipeline, add engine interface
- Implement sample engines (likely crypto)

Command Instructions

Instruction		Description
`icmd`	*eh*,*cmd*	Imperative command
`rcmd`	*eh*,*cmd*,*rd*	Receive-only command
`s1cmd`	*eh*,*cmd*,*rs1*	One-argument send-only command
`s2cmd`	*eh*,*cmd*,*rs1*,*rs2*	Two-argument send-only command
`s3cmd`	*eh*,*cmd*,*rs1*,*rs2*,*rs3*	Two-argument send-only command
`s1rcmd`	*eh*,*cmd*,*rd*,*rs1*	One-argument send-receive command
`s2rcmd`	*eh*,*cmd*,*rd*,*rs1*,*rs2*	Two-argument send-receive command

Command instructions issue commands to an engine bound to an engine handle. Imperative commands have no arguments; receive-only commands produce a result; send-only commands take only arguments; send-receive commands take arguments and produce a result. All command instructions can potentially produce fault traps.

Saving and Restoring Engines

Instruction		Description
Function		
`savelen`	*eh*,*rd*	Get state size
`save`	*eh*,*rd*,*rs*	Save state word at index *rs*
`restore`	*eh*,*rs1*,*rs2*	Restore word at index *rs2*
`savemem`	*eh*,*addr*,*imm*	Save whole state to memory
`restoremem`	*eh*,*addr*,*imm*	Restore whole state from memory

Engine states can be serialized either to registers or to memory using the save/restore instructions. Platforms may also implement secure save/restore functionality which transparently encrypts/decrypts engine states when saved and restored.

Virtualization

Instruction	Description
Function	
`transm` *trans*	Transition multiple
Perform transition *trans* for all engines owned at all lower privilege levels	
`saveevt` *addr*,*imm*	Save Engine Virtualization Table
Save enigine virtualization tables	
`loadevt` *addr*,*imm*	Load Engine Virtualization Table
Load enigine virtualization tables	

The **`transm`** is primarily used to pause and unpause engines when context-switching. Engine virtualization tables exist to map uniform virtual engine resources to possibly differing physical engine resources.

eric@metricspace.net