RISC-V Configuration Description

Version 0.1.0, January 5, 2022: This document is in Development state. Change should be expected.

Table of Contents

Preamble

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections):

- Abner Chang <abner.chang@hpe.com>, Hewlett Packard Enterprise
- Tim Newsome <tim@sifive.com>, SiFive, Inc

Chapter 1. Preface

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Chapter 2. Background

It is generally useful for software to be able to programmatically determine the capabilities of the hardware it is running on. Software needs to discover core and hart feature availability so it can take advantage of all of them. External debuggers need to know the same information so they can present an appropriate user interface. RISC-V extensions mostly use CSRs for this information, but that is not flexible enough for some kinds of data like hardware breakpoint capabilities, and cache hierarchy.

This document specifies syntax, semantics, discovery method, and access method for a static data structure that can accommodate implementation parameters of RISC-V standards beyond what can be easily put into CSRs. The structure is called a Configuration Structure (CS).

2.1. Use Cases

2.1.1. System Firmware

Typical system firmware is executed when the system is powered on. It initializes hardware and builds up firmware services or data structures for booting up the system to OS. Examples are U-Boot for embedded systems, and BIOS (majority firmware solution is UEFI TianoCore) for PCs and servers.

Through a combination of checking CSRs and accessing the system description, firmware can programmatically determine the hardware capabilities and configure hardware accordingly. These hardware capabilities can include availability and implemented features of Physical Memory Protection (PMP), Core Local Interrupt Controller (CLIC), Core Local Interruptor (CLINT), memory map, virtual memory, Trusted Execution Environment (TEE), and any future optional core and hart features.

The Configuration Structure is an efficient alternative to testing for specific hardware features (including handling failures) or customizing system firmware for the specific system it will run on.

Often system firmware will take the information it has learned from the Configuration Structure as well as through other methods, and encode it into a different industry-standard data structure like Devicetree, SMBIOS, or ACPI. This structure is then passed to the subsequent boot process.

2.1.2. External Debuggers

When an external debugger connects to a system, it would like to discover as much as possible about that system in as little time as possible. Some of this is merely to show the user (e.g. a manufacturer name), while other features are critical to the user (e.g. XLEN), and other features determine what kind of operations the user can perform (e.g. supported hardware trigger types). Most of these are already discoverable, although many require writing a value and checking the result to see whether support exists.

Any structure that's accessible from M-mode software will already be accessible by the debugger. There might be a structure embedded in the Debug Module itself which is only accessible by the debugger.

The debug feature that is most complex to describe is hardware triggers. Each hart may have billions of triggers (although 4 is more typical). Each of those triggers can be one of 4 types, and each type has its own options. Options are things like trigger on execute/load/store, in M/S/U mode, chain to other

trigger, exact/greater/less-than value match, etc. It's permissible for features to be implemented, but not in all combinations. E.g. greater value might work in combination with load/store, but not together with executed. Each trigger is configured by writing an XLEN-bit register.

In addition there are abstract commands, which have similar issues. There are a few commands, with a number of options.

2.2. Off-line Development Tools

A lot of development happens without access to the hardware, and software as well as hardware development tools can benefit from having a standardized description of hardware features to work from.

Chapter 3. Configuration Structure Schema

The Configuration Structure schema describes what implementation decisions can be put into the Configuration Structure, and how that data is encoded in the binary format. The schema is organized by RISC-V extension, and can itself be extended in a backwards-compatible way.

To accommodate minimal on-chip descriptions on small systems and larger descriptions for other use cases, implementation decisions are divided into the following three categories:

1. Primary information is only discoverable by reading the configuration structure, or by running a significant amount of code. Examples: the number of hardware triggers supported, ...
2. Secondary information is discoverable but not straightforward (e.g. WARL register). Examples: whether the F extension is supported, ...
3. Anything else goes in the Complete section. Examples: the value of XLEN, ...

Some extensions that only offer a handful of implementation decisions won't bother with this distinction, while for others it is important to accommodate primary descriptions for small systems.

The schema is written using ASN.1 basic notation and constraints. The schema itself is part of this specification, but resides separately in github, and includes all the files under github.com/riscv/configuration-structure/tree/master/schema. The top-level type is Top, defined in configuration-structure.asn.

3.1. Schema Best Practices

ASN.1 is extremely flexible. Below are some rules of thumb that impose limitations on its use that work best for the Configuration Structure use case:

1. All BOOLEANs should default to FALSE. This allows users to omit the value from their description and have it act as if it's set to FALSE explicitly. This does not affect the encoding at all. Example: `m BOOLEAN DEFAULT FALSE`
2. Constrain INTEGERs with an upper and lower bound if possible, if the upper bound is less than 256 times larger than the typical used value or the upper bound is less than 65536. INTEGERs that have no upper bound incur at least 1 byte in overhead when encoded. Example: `maskmax INTEGER (0..63)`
3. Constrain INTEGERs with a lower bound if possible. This can save a bit in the encoding. Example: `id INTEGER (0..MAX)`
4. Constrain SEQUENCE OF if they'll be small. Example: `single SEQUENCE SIZE(1..8) OF Integer3 OPTIONAL`
5. Add extension markers to your types unless you're really sure that we'll never want to add anything else to the type. If unused, it adds just 1 bit of overhead. Example: `...`
6. Define your types in such a way that it is hard or impossible to specify invalid configurations.

Chapter 4. Contents of Configuration Structure

The Configuration Structure contains a static description of a hardware platform, following the format descriped in the schema. It describes, in varying levels of detail, the implementation decisions made by the hardware designer. The description is static and is not affected by the current state of the system.

4.1. Human-Readable Format

ASN.1 defines a value syntax, but it's not well-supported among open source solutions. For now we'll use ASN.1 JER as the human-readable format for the content of Configuration Structure. JER is a JSON representation of the ASN.1 value. The Human-Readable format is backward compatible when new extensions are introduced to Configuration Structure schema.

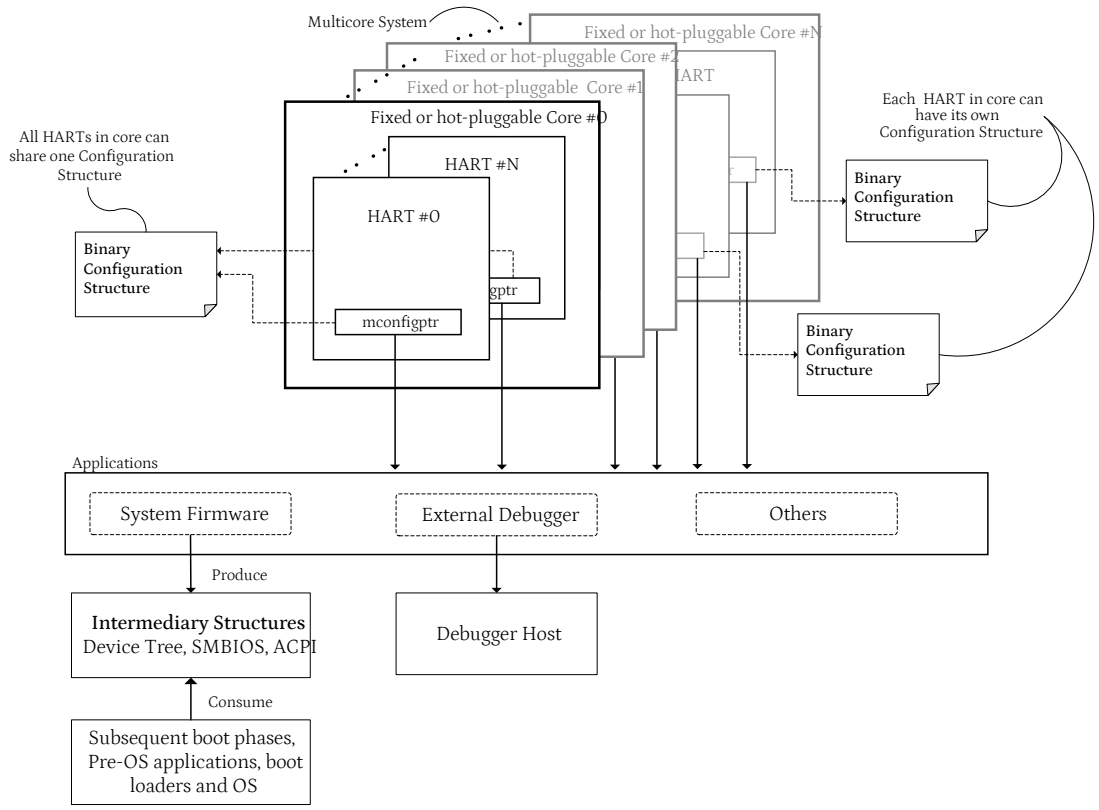
In the future, we should be able to accept YAML with little extra work, and the big immediate benefit of a format that supports comments.

4.2. Machine-Readable Format

The human-readable format is encoded to the binary using the standardized unaligned packed encoding rules (unaligned PER, see ASN.1 UPER, which is very compact. The binary format is backward compatible when new extensions are introduced into Configuration Structure schema.

Chapter 5. Access Method to Machine-Readable Format

The binary Configuration Structure is memory-mapped in system memory. mconfigptr contains the physical address where the structure starts. When software running on a hart wants to read the Configuration Structure, it reads mconfigptr, and then decodes the binary structure at that physical address. The structure will specify which parts apply to which harts (identified by hart ID), and the software can ignore any information that does not apply to the hart it's running on.



The above figure is a common example. Storage and provisioning of the Configuration Structure is implementation-specific and beyond the scope of this specification. There could be a single system-wide Configuration Structure, or a more complex arrangement for either single core or multicore systems.

5.1. Implementation Examples

There are several options for embedding the binary structure:

1. The structure can describe all harts, and be accessible over the memory bus. All harts have the same memory map and the same value in mconfigptr.
2. There might be several structures in the system, and different harts are pointed to different structures by having different pointers in mconfigptr.
3. There might be several structures in the system. Each hart has the same address in mconfigptr. The memory system provides a different configuration structure at that address depending on

which hart is performing the access.

4. A combination of 2 and 3 above could be used.

Hardware implementers have a lot of flexibility to handle everything from simple fixed systems to complex socketed systems. In each case it's straightforward to ensure that each hart can read a Configuration Structure that describes its own capabilities.

Chapter 6. External Industrial Standards

DeviceTree v0.3
ACPI v6.3
SMBIOS v3.5.0
RISC-V SMBIOS Type 44H

Index