

RgGen: An Open-Source CSR Automation Tool

Taichi Ishitani, PEZY Computing K.K., Tokyo, Japan (ishitani@pezy.co.jp)

Abstract—Control and status registers (CSRs) are essential components in SoC and FPGA designs, but manual CSR implementation is tedious and error-prone due to their sheer number and subtle variations. Moreover, preventing divergence between specifications and implementations is a persistent problem throughout development. This paper presents RgGen, an open-source CSR automation tool that accepts human-readable register map specifications and automatically generates RTL written in major HDLs and other required source files, eliminating these problems. RgGen supports multiple input formats and provides a rich set of pre-defined bit field types and register structures to handle a wide variety of CSR requirements.

Keywords—control and status register (CSR), register description, code generation, RTL, automation

I. INTRODUCTION

Control and status registers (CSRs) are essential components in SoC and FPGA designs. Each functional block typically has multiple CSRs that hold configuration values and status information, allowing the host processor to manage the block through bus accesses. As designs grow in complexity, the number of CSRs increases accordingly, and manual CSR implementation becomes a significant burden.

The problems of manual CSR implementation are threefold. First, the sheer volume of registers makes the work time-consuming. Second, because CSRs share similar but subtly different structures — varying in address, bit width, and bit position — coding errors are easy to make and hard to catch. Third, documentation updates are often deferred in favor of code changes, and the resulting divergence between specifications and their implementations will only grow over time, eventually rendering the specification unreliable.

Adopting a CSR automation tool is therefore essential to address these problems, and RgGen [1] is an open-source tool designed for this purpose, developed and continuously refined through more than a decade of production use. Given a human-readable register map specification, RgGen generates RTL, UVM Register Abstraction Layer (RAL) [2] models, C header files, and Markdown documentation in a single step, ensuring that all artifacts remain consistent with the specification at all times.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the features of RgGen along with its usage. Section IV presents use cases. Section V concludes the paper.

II. RELATED WORK

Several tools address CSR automation using different design philosophies. This section compares RgGen with SystemRDL [3] and IP-XACT [4], focusing on the underlying description principle and how each approach handles programmability.

Description Principle: Type vs. Property Composition. RgGen and SystemRDL, an Accellera standard for describing CSR register maps, follow different principles for writing a specification: selecting a named type versus composing a value from independent properties. RgGen applies the type-based principle throughout its specification model — not only for bit field access behavior (e.g., ‘rw’, ‘w1c’, ‘w1crs’) but also for register structures, where a type parameter selects among normal, array, indirect, and external forms (Section III.B). SystemRDL applies property composition just as broadly: bit field access behavior is described via properties such as ‘sw’, ‘hw’, ‘onread’, ‘onwrite’, and related properties, while other levels of the register hierarchy are likewise expressed through their own property sets. The following table summarizes the bit field case, where the contrast between the two principles is most concrete.

Table 1 Expressing bit field access behavior in RgGen and SystemRDL

Access Behavior	RgGen Type	SystemRDL Properties
Read-write, hardware read-only	rw	sw=rw, hw=r
Write-one-to-clear	w1c	sw=rw, onwrite=wocl, hwset=true
Write-one-to-clear + read-set	w1crs	sw=rw, onwrite=wocl, onread=rclr

Each approach carries trade-offs. RgGen's type-based model optimizes for quick recognition of common access patterns, at the cost of a closed vocabulary: a behavior outside the existing type vocabulary generally cannot be expressed. Many type names (e.g., 'rw', 'w1c') are already familiar abbreviations, but the vocabulary as a whole is not standardized across CSR automation tools the way SystemRDL's property model is. SystemRDL's property-combination model uses a set of orthogonal properties that can, in principle, express combinations beyond what a fixed type vocabulary anticipates, and its standardization enables shared semantics across independently developed tools. This compositionality is nonetheless bounded by the property vocabulary itself: a behavior with no corresponding property combination cannot be expressed at all. Indirect register access — where the value of one field determines which of several physical registers is accessed at a shared address, a pattern found in real designs such as UART CSRs — has no equivalent in SystemRDL's property set, whereas RgGen exposes it directly as its 'indirect' register type. Composition also comes at the cost of verbosity and a steeper learning curve for common cases, since the desired behavior must be composed correctly from multiple properties rather than selected by name.

Programmability. RgGen's register map file can also be written as a Ruby program as detailed in Section III.D. SystemRDL's specification defines ePerl, an embedded-Perl mechanism for generating procedural SystemRDL text, but this expansion happens as textual substitution before parsing and is not part of the language's semantic model.

The following table summarizes the comparison between RgGen and SystemRDL.

Table 2 Summary comparison of RgGen and SystemRDL

Aspect	RgGen	SystemRDL
Description principle	Named type	Composed from properties
Limitation from this principle	A behavior outside the type vocabulary cannot be expressed	A behavior with no property equivalent cannot be expressed (e.g., indirect register access)
Programmability	Ruby DSL, executed at runtime	ePerl, expanded as text before parsing
Standardization	Not standardized across CSR tools	Accellera standard

IP-XACT. IEEE 1685 (IP-XACT) is a widely adopted XML-based standard for packaging and integrating IP blocks, including memory-mapped register descriptions. Its XML representation and focus on tool interoperability for IP integration differ from RgGen's goal of a concise, human-writable specification with direct RTL/RAL/documentation generation. RgGen does not currently support IP-XACT as an output format; adding it is planned future work.

III. RGEN: FEATURES AND USAGE

A. Features

RgGen provides the following key features.

- **Bit Field and Register Structure:** RgGen provides 44 pre-defined bit field types and supports non-contiguous bit field assignments, covering a wide variety of access behaviors. For register structures, RgGen supports normal, array, indirect, and external types to handle diverse CSR requirements. Details are described in Section III.B.
- **Input Formats:** RgGen accepts register map specifications in formats familiar to most engineers: structured text formats (YAML, JSON, TOML), spreadsheet formats (XLSX, ODS, CSV), and SystemRDL via a dedicated plugin. For more programmatic descriptions, a Ruby [5]-based domain-specific language (DSL) is also supported, allowing register maps to be written using any Ruby construct.

- **Output Files:** From a single register map specification, RgGen generates RTL, a UVM RAL model, a C header file, and Markdown documentation. The RTL is generated not only in major HDLs (SystemVerilog, Verilog, and VHDL) but also in Veryl [6], and supports industry-standard bus protocols, including AMBA® APB, AMBA® AXI4-Lite, Avalon®-MM, and Wishbone.
- **Extensibility:** RgGen supports a plugin mechanism that allows users to extend RgGen to suit their own environment. Examples include adding custom bit field types to handle project-specific access behaviors, adding custom bus protocol adapters to support proprietary bus interfaces, and adding new output generators to produce additional file formats.
- **Browser-Based Editor:** For users who prefer a GUI, RgGen also provides a browser-based register map editor [7] that runs entirely client-side via a WebAssembly build of RgGen, requiring no server or installation.

B. Bit Field and Register Structure

Bit Field Types. RgGen provides 44 pre-defined bit field types covering a wide range of bit field access behaviors. Simple types such as read-write ('rw') and read-only ('ro') handle common cases, while more complex types such as write-one-to-clear ('w1c') and write-one-to-clear-read-set ('w1crs') support interrupt-related patterns. Users specify the desired behavior simply by setting the type parameter in the register map.

Register Structures. RgGen supports not only normal registers, but also array, indirect, and external types. Array registers allow multiple instances of the same register structure to be defined compactly. Indirect registers use the value of another bit field as an access qualifier, enabling multiple registers to share the same offset address. External registers provide an interface port for connecting external modules such as other register blocks or SRAMs.

C. Input Files

RgGen requires two types of input files: a configuration file and a register map file. The configuration file specifies common settings shared across all register blocks. The register map file describes CSR specifications in detail.

The configuration file is written as a mapping in YAML, JSON, or TOML format. The following is an example in YAML:

```
bus_width: 32
address_width: 16
protocol: axi4lite
```

Figure 1. Example of configuration file

The key fields are as follows:

Table 3. Key fields of configuration file

Field Name	Description
bus_width	Data bus width in bits
address_width	Address width in bits
protocol	Bus protocol for the generated RTL

The register map has a four-level hierarchy: bit field, register, register file, and register block. The bit field is the smallest unit, occupying a specific bit range and holding a single configuration value or status information. A register groups one or more bit fields and is assigned a unique address. The register file is an optional level that provides a virtual sub-grouping of registers. The register block is the top-level container that groups all registers belonging to a functional block. This hierarchy directly mirrors the structure of actual CSR blocks in hardware, making the register map straightforward to write and easy to understand.

The following is an example register map written in the Ruby DSL:

```

register_block {
  name 'block_0'
  byte_size 256

  register {
    name 'register_0'
    offset_address 0x00
    bit_field { name 'bit_field_0'; bit_assignment lsb: 0, width: 4; type :rw; initial_value 0 }
    bit_field { name 'bit_field_1'; bit_assignment lsb: 4, width: 2; type :w1; initial_value 0 }
  }

  register {
    name 'register_1'
    offset_address 0x08
    bit_field { name 'bit_field_0'; bit_assignment lsb: 0, width: 4; type :ro }
    bit_field { name 'bit_field_1'; bit_assignment lsb: 8, width: 8; type :rof; initial_value 0xab }
  }
}

```

Figure 2. Example of register map

In the Ruby DSL, each level of the hierarchy is represented as a block enclosed in braces, with inner levels nested within outer ones. A ‘`register_block {}`’ block defines a register block and contains ‘`register {}`’ blocks. Each ‘`register {}`’ block defines a register and contains ‘`bit_field {}`’ blocks. A ‘`bit_field {}`’ block defines an individual bit field.

The key fields at each level are summarized below:

Table 4. Key fields of register map

Field Name	Description
Register Block	
name	Register block name
byte_size	Address space size in bytes
Register	
name	Register name
offset_address	Byte offset address
type	Register type (normal if omitted)
Bit Field	
name	Bit field name
bit_assignment	Bit position (lsb) and width (width)
type	Bit field type
initial_value	Reset value

D. Programmability via the Ruby DSL

Dedicated CSR automation tools typically accept a declarative, structured format — for example, Cheby [8] uses YAML, PeakRDL [9] uses SystemRDL, and reggen [10] uses Hjson. RgGen's DSL, by contrast, is built on Ruby, so any Ruby construct can be used directly in the register map. Accepting a general-purpose programming language for the specification itself is uncommon among these tools. It provides the following advantages:

- **Expressions** allow values such as field widths, addresses, and names to be computed rather than hardcoded.
- **Conditionals** enable a single register map file to cover architecture-dependent variations without duplication.
- **Loops** allow large repetitive register structures to be defined concisely in just a few lines, eliminating the need to manually enumerate each register.

The following example, taken from a RISC-V CSR implementation [11], demonstrates the first capability: the register block name is constructed using string interpolation, and the width of the ‘base’ bit field is computed as

‘xlen-2’, where ‘xlen’ is the integer register width read from the configuration file via ‘configuration.bus_width’.

```
xlen = configuration.bus_width

register_block {
  name "riscv_csr_m_level_xlen#{xlen}"

  register {
    name 'mtvec'
    offset_address 0x305
    bit_field { name 'mode'; bit_assignment lsb: 0, width: 2; type :rw; initial_value 0 }
    bit_field { name 'base'; bit_assignment lsb: 2, width: xlen - 2; type :rw; initial_value 0 }
  }
}
```

Figure 3. Register map using Ruby expressions for architecture-dependent definitions

With this single register map file, RgGen generates correct RTL for both RV32 and RV64 depending on the ‘bus_width’ value in the configuration file.

IV. USE CASES

RgGen has been adopted in production at PEZY Computing to manage all CSRs in our designs. The designs contain more than 30 register blocks in total, comprising 606 registers, all of which are described in YAML and processed by RgGen. RgGen generates more than 50,000 lines of RTL code from these register maps. While auto-generated code is typically more verbose than equivalent hand-written RTL, this scale illustrates the burden that manual implementation and maintenance of these register blocks would impose.

The register maps make use of the full range of RgGen's structural features, including array registers, indirect registers, and non-contiguous bit field assignments. This demonstrates that RgGen is capable of handling the complexity found in real production designs.

RgGen generates three types of output from the register maps. Veril RTL is used as the implementation of the CSR blocks. UVM RAL models are integrated into the verification testbench and used across the regression suite. For documentation, Markdown output is automatically published as a documentation page on every commit. Figure 4 shows a screenshot of the generated documentation page. This ensures that specifications and implementations remain permanently synchronized — directly addressing the divergence problem described in Section I.

A custom plugin was developed to adapt RgGen to the project's specific requirements. This includes support for the custom bus protocol, additional bit field types to handle project-specific access behaviors, and extended Markdown output items to meet internal documentation standards. This demonstrates that RgGen's plugin mechanism is practical for real-world customization needs.

cfg_dmac

- byte_size
 - = 1024
- bus_width
 - = 32
- base_addresses:
 - base_address: 0x0108000
 - base_address: 0x0108400
 - base_address: 0x0108800
 - base_address: 0x0108c00
 - base_address: 0x0109000
 - base_address: 0x0109400
 - base_address: 0x0109800
 - base_address: 0x0109c00
- dmac_control
 - offset_address
 - = 0x000
 - type
 - = default

name	bit_assignments	type	initial_value	reference	labels	comment
run	[0]	wrtng				1 書き込みで転送開始
stop	[16]	wrtng				1 書き込みで強制停止
clear_debug_info	[24]	wrtng				1 書き込みでデバッグ情報を初期化
busy	[31]	ro	0x0			1: DMA 実行中

Figure 4. Documentation page generated from Markdown

RgGen has also been adopted in several open-source projects [12][13], further demonstrating its versatility and maturity as a production-ready tool.

V. CONCLUSION

RgGen addresses the problems of manual CSR implementation by generating RTL and other required source files from human-readable register map specifications. Its rich feature set covers a wide variety of CSR requirements, and its plugin mechanism allows users to extend RgGen to suit their own environment. RgGen has been adopted in production at PEZY Computing and in several open-source projects, demonstrating that it is a practical and mature tool ready for use in real-world designs.

ACKNOWLEDGMENT

The author would like to thank all users and contributors of RgGen for their valuable feedback and continued support.

REFERENCES

- [1] T. Ishitani, "RgGen". Available online: <https://github.com/rggen/rggen> (accessed on 16 August 2026).
- [2] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017), vol., no., pp.1-458, 14 Sept. 2020, doi: 10.1109/IEEESTD.2020.9195920.
- [3] Accellera Systems Initiative. SystemRDL 2.0: Register Description Language. Available online: <https://www.accellera.org/downloads/standards/systemrdl> 2018 (accessed on 6 August 2026).
- [4] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," in IEEE Std 1685-2022 (Revision of IEEE Std 1685-2014), vol., no., pp.1-750, 28 Feb. 2023, doi: 10.1109/IEEESTD.2023.10054520.
- [5] "Ruby Programming Language". Available online: <https://www.ruby-lang.org> (accessed on 16 August 2026).
- [6] N. Hatta and The Veryl contributors, "Veryl Hardware Description Language". Zenodo, May 01, 2026. doi: 10.5281/zenodo.19934617.
- [7] T. Ishitani, "RgGen Web UI". Available online: <https://rggen.github.io/rggen-webui/> (accessed on 16 August 2026).
- [8] "Cheby". Available online: <https://gitlab.cern.ch/be-cem-edl/common/cheby> (accessed on 16 August 2026).
- [9] A. Mykyta, "Peakrdl". Available online: <https://github.com/SystemRDL/PeakRDL> (accessed on 16 August 2026).
- [10] OpenTitan Project, "reggen". Available online: <https://opentitan.org/book/util/reggen/index.html> (accessed on 16 August 2026).
- [11] A. Waterman and K. Asanović (Eds.), "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," RISC-V International. Available online: <https://github.com/riscv/riscv-isa-manual> (accessed on 16 August 2026)
- [12] A. Ignacio, "axi_dma". Available online: https://github.com/aignacio/axi_dma (accessed on 16 August 2026)
- [13] Matthias, "WdgRV". Available online: <https://github.com/matztron/WdgRV> (accessed on 16 August 2026)