



RgGen: An Open-Source CSR Automation Tool

Taichi Ishitani
PEZY Computing K.K.

Agenda

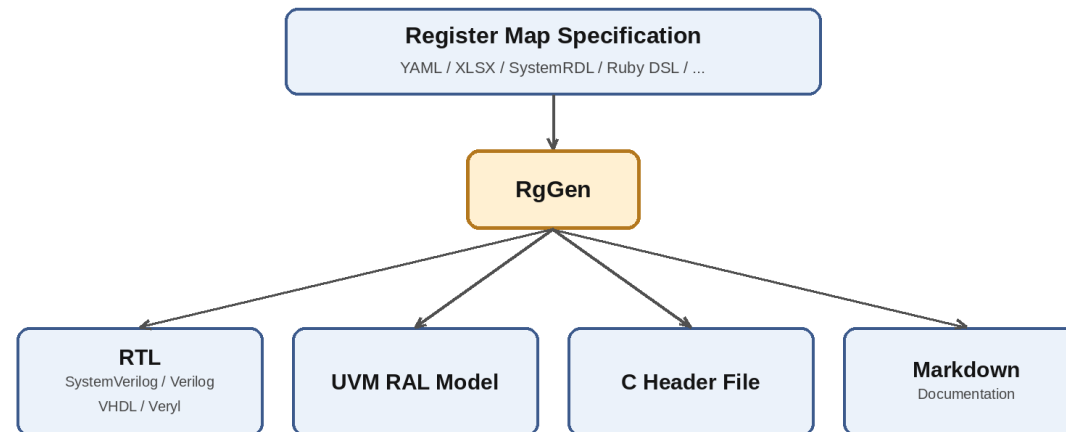
- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- Use Case at PEZY Computing
- Conclusion

Agenda

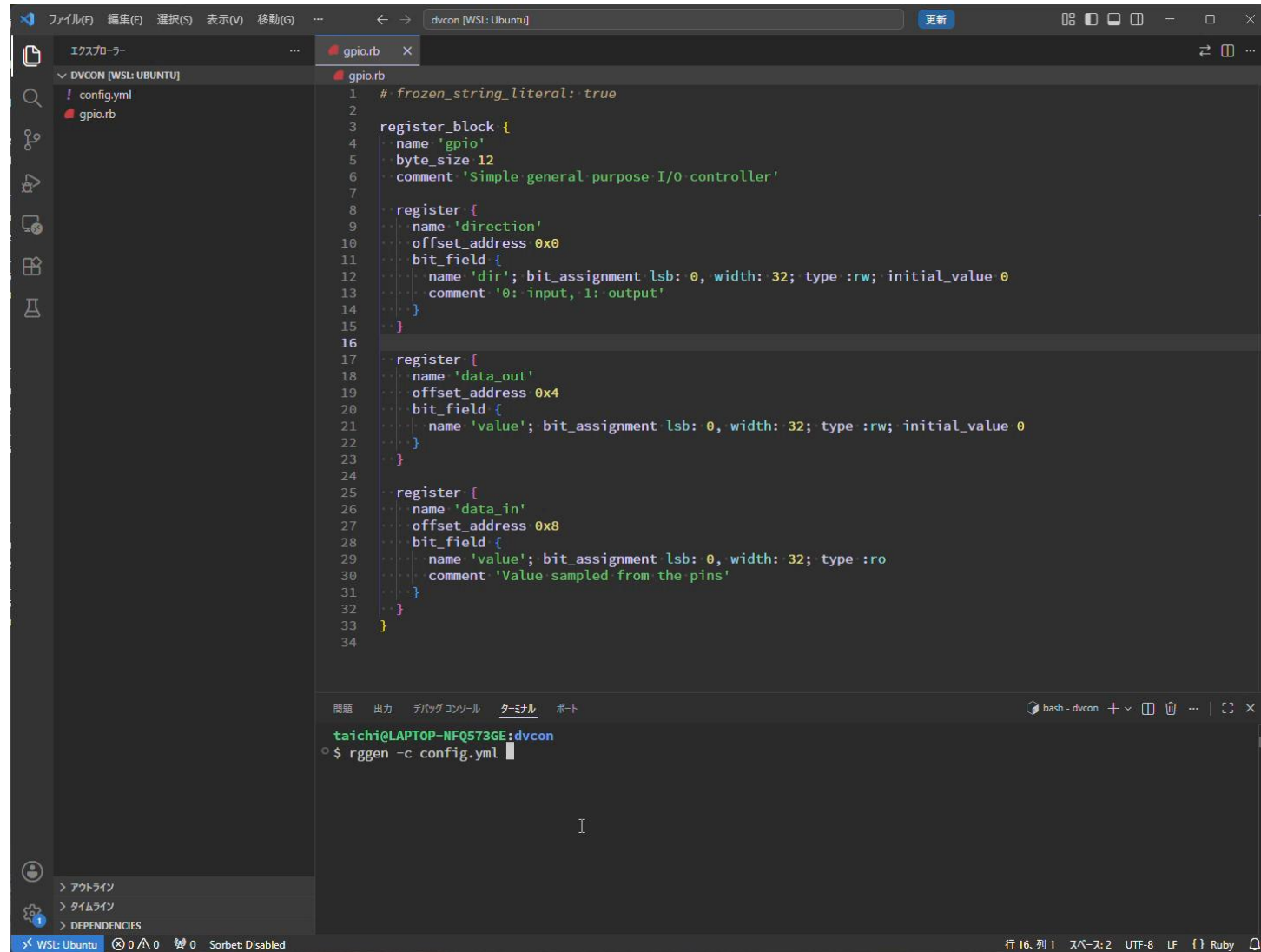
- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- Use Case at PEZY Computing
- Conclusion

Quick Introduction

- RgGen: Open-source CSR automation tool
 - Generate RTL, UVM RAL models, C header files, and Markdown documentation from human-readable specifications
 - Developed and continuously refined through more than a decade of production use
 - Available on GitHub: <https://github.com/rggen/rggen>



Demo



The screenshot shows a VS Code editor window with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'DVCON [WSL: UBUNTU]' containing 'config.yml' and 'gpio.rb'. The 'gpio.rb' file is open in the editor, showing Ruby code for a hardware register block. The code defines a 'gpio' register block with a 'direction' register and two data registers ('data_out' and 'data_in'). The terminal window shows the command 'rggen -c config.yml' being executed.

```
1 # frozen_string_literal: true
2
3 register_block {
4   name 'gpio'
5   byte_size 12
6   comment 'Simple general purpose I/O controller'
7
8   register {
9     name 'direction'
10    offset_address 0x0
11    bit_field {
12      name 'dir'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0
13      comment '0: input, 1: output'
14    }
15  }
16
17   register {
18     name 'data_out'
19     offset_address 0x4
20     bit_field {
21       name 'value'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0
22     }
23   }
24
25   register {
26     name 'data_in'
27     offset_address 0x8
28     bit_field {
29       name 'value'; bit_assignment lsb: 0, width: 32; type :ro
30       comment 'Value sampled from the pins'
31     }
32   }
33 }
34
```

Terminal output:

```
taichi@LAPTOP-NFQ573GE:dvcon
$ rggen -c config.yml
```

Demo

The screenshot shows the RgGen web interface in a browser. The URL is <https://rggen.github.io/rggen-webui/>. The interface includes a top navigation bar with links to GitHub, Wiki, and Versions, and buttons for Load Sample, Reset, Preview YAML, Integration Guide, Help, Download YAML (ZIP), and Generate (ZIP).

The main configuration area is divided into two sections: CONFIG and BLOCK. The CONFIG section shows Bus Width set to 32, Address Width set to 16, and Protocol set to apb. The BLOCK section shows a block named block_0 with Bus Width set to (use config) and Byte Size set to 256. Below the block configuration, there is a table for registers.

Name	Offset	Type	Comment
reg_0	0x	(default)	comment

Below the register table, there is a table for bit fields.

Name	LSB	Width	Type	Init	Comment
Adv	auto	1	rw	0	comment

At the bottom of the interface, there are buttons for + Add Register, Expand All, and Collapse All. A right sidebar contains a HELP section with tabs for Config, Block, Register, and Bitfield. The Config tab is active, showing information about Bus Width, Address Width, and Protocol.

Bus Width
Data bus width in bits. All registers in the project use this width unless overridden per block. Must be 8, 16, 32, or 64.

Address Width
Number of address bits. Determines the addressable range of the register map.

Protocol
Bus interface protocol used for the generated RTL. Available options: apb, axi4lite, avalon, wishbone.

Agenda

- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- Use Case at PEZY Computing
- Conclusion

Problem with Manual CSR Implementation

- Manual implementation suffers from three problems:
 1. Volume: simply time-consuming
 2. Errors: address, width, and bit position vary subtly across many registers, easy to get wrong
 3. Divergence: spec updates lag behind code changes, drifting further from the implementation over time
- Solution: Adopting a CSR automation tool
 - RgGen is designed for exactly this purpose

Agenda

- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- **Features**
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- Use Case at PEZY Computing
- Conclusion

Overview

- Bit Field & Register Types
 - 44 pre-defined bit field types
 - normal/array/indirect/external register types
- Input Formats
 - Structured Text (YAML/JSON/TOML)
 - Spreadsheet (Excel/LibreOffice/CSV)
 - SystemRDL
 - Ruby DSL
- Output Files
 - RTL
SystemVerilog/Verilog/VHDL/Veryl
 - UVM RAL models
 - C header files
 - Markdown documentation
- Extensibility
 - Plugins for custom bit field types, bus protocols, and output generators
- Browser-Based Editor
 - Edit register maps and generate outputs directly in the browser
 - <https://rggen.github.io/rggen-webui/>

Register Map: Hierarchy & Example

```
register_block {  
  name 'gpio'  
  byte_size 12  
  comment 'Simple general purpose I/O controller'  
  
  register {  
    name 'direction'  
    offset_address 0x0  
    bit_field {  
      name 'dir'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0  
      comment '0: input, 1: output'  
    }  
  }  
}  
  
register {  
  name 'data_out'  
  offset_address 0x4  
  bit_field {  
    name 'value'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0  
  }  
}  
  
register {  
  name 'data_in'  
  offset_address 0x8  
  bit_field {  
    name 'value'; bit_assignment lsb: 0, width: 32; type :ro  
    comment 'Value sampled from the pins'  
  }  
}  
}
```

Define register block layer

Define register layer

Define field layer

- 4 level structures:
 - Register block
the top-level container for a functional block
 - Register file (optional)
a virtual sub-grouping of registers
 - Register
one or more bit fields at a fixed address
 - Bit field
a bit range holding one value/status (smallest unit)

Register Map: Bit Field & Register Types

- RgGen provides both simple and complex types
 - Covering real-world design needs without trouble
- Bit field types: named types select access behavior
 - simple (rw, ro, ...)
 - complex (w1c, w1crs, ...)
- Register structures
 - Normal
fixed address
 - Array
repeated structure, indexed
 - Indirect
shared address, selected by another field's value
 - External
interface port to an external module

Programmability via the Ruby DSL

- RgGen's register map can be written in Ruby
 - Expressions, conditionals, and loops work directly, right in the register map
- This is an uncommon capability
 - Other CSR automation tools use a declarative format
 - Cheby: YAML
 - PeakRDL: SystemRDL
 - reggen: Hjson

Programmability via the Ruby DSL

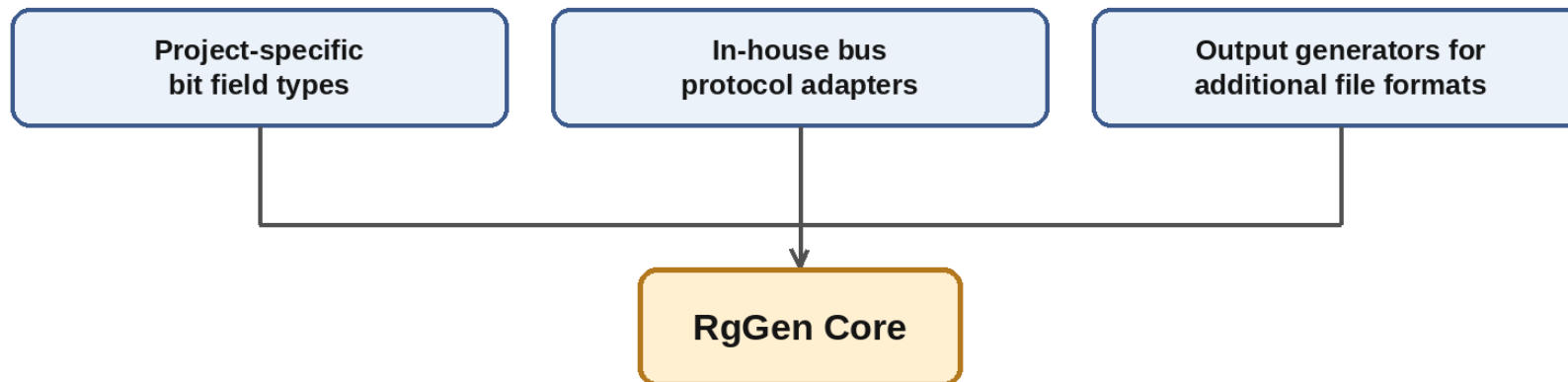
- Example 1
 - RISC-V CSR 'mtvec' register
 - Width computed from 'xlen'
- Example 2
 - RISC-V CSR 'mcycleh' register
 - Exists only for RV32 architecture

```
# Example 1 (expression)
register {
  name 'mtvec'
  offset_address byte_address(0x305)
  bit_field {
    name 'mode'
    bit_assignment lsb: 0, width: 1; type :rw; initial_value 0
  }
  bit_field {
    name 'base'
    bit_assignment lsb: 2, width: xlen - 2; type :rw; initial_value 0
  }
}

# Example 2 (condition)
if xlen == 32
  register {
    name 'mcycleh'
    offset_address byte_address(0xB80)
    bit_field {
      bit_assignment lsb: 0, width: xlen; type :counter; initial_value 0
      reference 'mcountinhibit.cy'
    }
  }
end
```

Extensibility: Plugin Mechanism

- RgGen can be extended by building plugins
 - Add features not covered by the built-in functionality:
 - Add project-specific bit field types
 - Add in-house bus protocol adapters
 - Add output generators for additional file formats
- Already used in production at PEZY Computing



Agenda

- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- **FAQ: Why Not Standardized SystemRDL?**
- Use Case at PEZY Computing
- Conclusion

FAQ: Why Not Standardized SystemRDL?

- SystemRDL
 - Accellera standard for describing CSR register maps
 - Supported by both open-source and commercial tools
 - [PeakRDL](#)
 - [Agnisys IDesignSpec](#)
- A common question: "Why not use SystemRDL?"
 - Differences in design principle and expressiveness

FAQ: Why Not Standardized SystemRDL?

RgGen

```
register_block {
  name 'gpio'
  byte_size 12
  comment 'Simple general purpose I/O controller'

  register {
    name 'direction'
    offset_address 0x0
    bit_field {
      name 'dir'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0
      comment '0: input, 1: output'
    }
  }

  register {
    name 'data_out'
    offset_address 0x4
    bit_field {
      name 'value'; bit_assignment lsb: 0, width: 32; type :rw; initial_value 0
    }
  }

  register {
    name 'data_in'
    offset_address 0x8
    bit_field {
      name 'value'; bit_assignment lsb: 0, width: 32; type :ro
      comment 'Value sampled from the pins'
    }
  }
}
```

SystemRDL

```
addrmap gpio {
  default precedence = hw;

  name = "GPIO";
  desc = "Simple general purpose I/O controller";

  // Most fields are software programmable and read by hardware
  default sw = rw;
  default hw = r;

  reg {
    name = "Port Direction";
    field {
      desc = "0: input, 1: output";
      reset = 0x0;
    } dir[32];
  } direction;

  reg {
    name = "Output Data";
    field { reset = 0x0; } value[32];
  } data_out;

  reg {
    name = "Input Data";
    field {
      desc = "Value sampled from the pins";
      sw = r; hw = w;
    } value[32];
  } data_in;
};
```

FAQ: Why Not Standardized SystemRDL?

- Different design principle
 - RgGen
 - Named types (rw, w1c, ...)
 - Readable, but a closed vocabulary
 - SystemRDL
 - Composes values from properties such as sw/hw/onread/onwrite
 - Flexible composition, but verbose and a steeper learning curve
- Expressiveness gap
 - SystemRDL cannot express behaviors outside its property set (e.g., indirect registers, common in UART CSRs)
- Standards compliance alone isn't enough
 - Tools grounded in real-world requirements matter too

Agenda

- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- **Use Case at PEZY Computing**
- Conclusion

Use Case at PEZY Computing

- All CSR blocks managed with RgGen
 - 30+ register blocks, 600+ registers
 - Written in YAML → equivalent to 50,000+ lines of generated RTL
- Full use of structural features
 - array registers, indirect registers, non-contiguous bit field assignments
- Custom plugin for project-specific needs:
 - In-house bus protocol adapter
 - Additional bit field types
 - Extended Markdown output

Use Case at PEZY Computing

- RgGen generates three outputs from the same register map:
 - Veryl RTL
 - UVM RAL
 - Integrated into the verification testbench, used across the regression suite
 - Markdown
 - Documentation page automatically updated on every commit
- Avoids the problems of manual CSR implementation
 - Far less effort for changes
A single edit to the register map regenerates all three outputs
 - Divergence prevented
Specifications and implementation stay in sync

Agenda

- Quick Introduction
 - Demo
- Problem with Manual CSR Implementation
- Features
 - Overview
 - Register Map
 - Programmability via the Ruby DSL
 - Extensibility
- FAQ: Why Not Standardized SystemRDL?
- Use Case at PEZY Computing
- Conclusion

Conclusion

- RgGen automatically generates RTL/RAL/C header/documentation from human-readable specifications
 - Solving the problems of manual CSR implementation
 - Available on GitHub: <https://github.com/rggen/rggen>
- A mature, production-ready tool, refined over a decade of real-world use
- Proven in production at PEZY Computing

Thank you!