RDL - Register Description Language

Julian Gorfajn

Maxtor Corporation

Julian_Gorfajn@maxtor.com

ABSTRACT

In order to reuse designs between projects, we often need to modify hardware interfaces, software interfaces, and documentation to meet the standards of the new project. One common aspect of this is change to the processor interface of the design. This could include a change to the bus interface, a change in language choice by firmware, or a different documentation style.

This paper presents a new language called the Register Description Language (RDL). The language is designed to contain information necessary to automatically generate a RTL processor interface, header files, documentation, and the possibility of other outputs. The paper will show how changes in rules on these various generated outputs can be done without the need to modify the design.

Table of Contents

Table of Figures

1.0 Introduction – What is a Register Description Language?

1.1 Concept

With almost every design change, a firmware-accessible register change follows suit. This register change not only causes an RTL change – typically there are specification changes and firmware changes to match. Since all of these changes are done independently, there is a high likelihood of error, which results in a schedule impact.

Imagine if you could describe registers in a language that facilitates automatic generation tools for all these different outputs (specification, RTL, firmware headers files) — a Register Description Language.

1.2 History

In 1997, in order to handle a design that supported two different processor interfaces, RDL1.0 was created.

```
SECTIONNAME=Section Name Here;
BASEADDR=0x10000;

reg1[15:0]@0x00:{0[1:0],bitf3[10:0],bitf2,bitf1[1:0]}=(my register
description);
bitf1[1:0]:{0,rw}=(This is bitfield 1);
bitf2:{1,ro}=(This is bitfield 2);
bitf3[10:0]:{0,punch}=(This is bitfield 3);
```

Figure 1 - RDL1.0

The RDL1.0 language consists of three types of lines:

- Global Directives: These (like SECTIONNAME and BASEADDR) are global to all registers.
- Register Description: This shows the size, address, description, and bitfields contained in the register. (The number 0 represents unimplemented bits.)
- Bitfield Description: This shows the reset state, type and description of the bitfield.

While RDL1.0 is functional, it isn't extensible. For example, there is no way to implement enumerations for bitfields. Also, type information (reset, etc) is located in a comma-delimited field that is not very descriptive.

2.0 RDL 2.0

In 2000, the decision was made to develop a new language based on lessons learned from RDL1.0. The key issue of RDL1.0 was lack of extensibility. RDL2.0 resolved this issue by allowing an arbitrary number of attributes to be attached to any register or bitfield.

```
//this is a comment

/*this is also a comment*/

map top[31:0]@0x10000 {
  title("My Registers");
  reset(0);
  fwaccess(rw);
  hwaccess(ro);

  reg config[7:0]@0b0100 {
    title("Configuration");
    desc("This is a configuration register.");
    bit start@0 {
      hwaccess(rw);
      fwaccess(punch);
    }
  }

  reg address[15:0]@6 {
    desc("This is an address register.");
    unsigned addr[12:0]@0;
  }
}
```

Figure 2 - RDL2.0

2.1 Elements

Elements are the data type of RDL2.0. They separate address space into unused and used areas. Currently there are four types of elements: `maps`, `registers`, `bitfields`, and `arrays`. Elements look like C structs. Figure 2 shows the following elements: `top`, `config`, `start`, `address`, and `addr`. Element definitions are either followed by braces that contain sub-elements and functions, or terminated by a semicolon (e.g. `addr`). All elements have two attributes: size and address. The size is represented by a low and high index similar to Verilog. Size information is element specific (e.g. `reg` elements have size in bits where `map` elements have size in bytes), Again, as with Verilog, elements of size 1 do not need to specify their size (e.g. start). The address is specified after an @ sign. Addresses can be listed as hex, decimal or binary (e.g. 0b100).

2.1.1 The `map` Element

The `map` element is used to group a set of registers. In Figure 2, top is a map element. `Top` is a 32-byte map starting at address 0x10000. Map element sizes are always in bytes. Registers and bitfields described inside maps inherit features of the map. All register addresses are relative to the map address. Bitfield attributes such as reset, firmware access, and hardware access can be defined in the map as defaults and overwritten later. Address regions in the map that are not defined by registers are considered reserved. Generators can assume that these areas are not accessed and optimize accordingly.

Maps can contain maps and registers. By placing maps inside other maps, you are able to group registers and have a common set of defaults.

2.1.2 The `reg` Element

The `reg` element is used to define registers. In Figure 2, `config` is a `reg` element. `Config` is an 8-bit register at address 4 relative to the `map`, `top`. Thus, `config` is at address 0x10004. `Reg` element sizes are always in bits, and must have widths of the form 2^n where $n \geq 3$ (e.g. 8, 16, 32, etc) and must have addresses that are aligned to their size. That is – an 8-bit register can be anywhere, a 16-bit register must have an address that is divisible by 2 and a 32-bit register must have an address that is divisible by 4. Bit addresses in a `reg` that are not defined by bitfield are considered reserved. Generators can assume that these areas are not accessed and optimize accordingly.

2.1.3 Bitfield Elements

There are currently four types of bitfields: `bit`, `signed`, `unsigned` and `reserved`. `Bit` is the generic bit type (e.g. `start` in Figure 2). Use bit when nothing else applies. `Signed` and `unsigned` are used to represent multi-bit numeric values (e.g. `addr`). `Reserved` is a special bitfield where you wish to implement a bitfield (that is – there will be flops in the design) however, you don't have a defined function for them yet. RTL generators should treat `reserved` bitfields as real registers and not optimized them away.

2.1.4 The `array` Element

The `array` element is used to create multiple registers with the same attributes.

```
array loop[127:0]@4 {
  reg entry[15:0]@0 {
    unsigned stuff[10:0]@0;
  }
}
```

Figure 3 – Array Element

In Figure 3, we have 128 registers. Each register (`entry`) is 16 bits wide, and the bitfield they contain (`stuff`) is 11 bits wide. The size in the array element is based on the number of elements it contains. In this case – there are 128 entries. All the other rules must still apply. Since the contents of this `array` are 16-bit registers, the array must be 16-bit aligned.

2.2 Functions

```
map top[31:0]@0x10000 {
  title("My Registers");
  reset(0);
  fwaccess(rw);
  hwaccess(ro);

  reg config[7:0]@4 {
    title("Configuration");
    desc("This is a configuration register.");
    bit start@0 {
      hwaccess(rw);
      fwaccess(punch);
    }
    bit stop@1;
  /* more registers here */
}
```

Figure 4 - Functions

While elements define which registers and bitfields exist, *functions* describe them. In Figure 4 we see a number of functions (all in *italics*). Some functions are used for documentation (e.g. title()). Others describe the functionality of a bitfield (e.g. reset()). Some functions may be defined and then overwritten. For example, reset(), fwaccess() and hwaccess() are all bitfield functions. They are listed in the `map`, but then some are overwritten in the definition of the `bit` – `start`. This means that by default, all bitfields in the `map top` inherit a reset(0), fwaccess(rw) and hwaccess(ro). `Start` keeps the default reset(0) but overrides the other two.

2.2.1 Bitfield Description Functions

Five different functions may be used to describe the specifics of a bitfield.

reset(arg)

The reset() function describes the reset state of the bitfield. Any numerical argument is valid (hex, decimal or binary) as well as X meaning unknown reset state. If the argument is larger than the width of the bitfield, the lower bits of the argument will be used. If the argument is smaller than the bitfield, then it is zero-extended. This is useful when a reset() value is set as a default.

resettype(arg)

The resettype() function allows for setting a range of reset causes. RTL generators will use this argument as the reset source for the flop.

hwaccess(arg)

The hwaccess() function specifies what kind of access hardware will have to the bitfield. There are two valid arguments: ro (read only) and rw (read/write).

fwaccess(arg)

The fwaccess() function specifies what kind of access firmware will have to the bitfield. There are four valid arguments as described in Figure 5.

Value	Name	Description
ro	Read only	Firmware can only read the bitfield. (This value requires hwaccess(rw).)
rw	Read/Write	Firmware can read and write the bitfield.
w1tc	Write one to clear	Firmware can read the bitfield. The bitfield is cleared when writing a 1 to it. (This value requires hwaccess(rw).)
punch	Punch	Setting this bitfield holds state. When hardware sees the bit is set, it may start a machine and at some point hardware will clear the bit. (This value requires hwaccess(rw).)
wo	Write only	Firmware can write but not read the bitfield.

Figure 5 - fwaccess() values

value((arg,arg,arg)…)

```
value(
      (SIGHUP,  1, “Hangup detected on controlling terminal or
death of controlling process”),
      (SIGINT,  2, “Interrupt from keyboard”),
      (SIGQUIT, 3, “Quit from keyboard”),
      (SIGILL,  4, “Illegal Instruction”),
      (SIGABRT, 6, “Abort signal from abort(3) ”),
      (SIGFPE,  8, “Floating point exception”),
      (SIGKILL, 9, “Kill signal”),
      (SIGSEGV,11, “Invalid memory reference”),
      (SIGPIPE,13, “Broken pipe: write to pipe with no readers”),
      (SIGALRM,14, “Timer signal from alarm(2)”),
      (SIGTERM,15, “Termination signal”)
      );
```

Figure 6 - value()

The value() function is used to describe bitfields which are enumerated types. In Figure 6, an enumerated type is defined.[1] Value() takes a comma-delimited list of triplets. The first value is the enumerated mnemonic. The second value is the numerical value of that mnemonic and the third value is a description. The numerical value can be decimal, hexadecimal, or binary. Unspecified numeric values are considered reserved. Generators can assume that these are not accessed and optimize accordingly.

[1] The enumerated type listed is standard POSIX names and values for signals.

2.2.2 Documentation Functions

Documentation functions provide more details about your registers and bitfields. Multiple generators could use this information in comments to make the code clearer. Specification generators make heavy use of these functions.

title(arg)

The title() function is used to give a name to the register or bitfield. If the element does not have a title() function, then the instance name is used. This function should only be used if the name is clearer.

desc(arg,…)

The desc() function is used to give a description of the register or bitfield. It can take a comma-delimited list of strings, ref() functions, and para() functions.

ref(arg)

```
map map1[10:0]@0 {
  reg reg1[7:0]@0 {
    bit bit1@0;
  }
  reg reg2[7:0]@1 {
    bit bit1@0;
    bit bit2@1 {
      desc("We can cross reference everywhere. There is the local
bit1 which is ", ref(bit1), " and the distant bit1 which is ",
ref(reg1.bit1), ". We can also reference registers such as reg1
like this ", ref(reg1));
    }
  }
}
```

Figure 7 - Using ref()

The ref() function is used to create a cross reference in desc(). The cross reference is a hierarchical register or bitfield name done similar to Verilog hierarchical names.

para()

The para() function denotes the end of paragraphs in your desc(). Generators will be able to use this to separate text into separate paragraphs.

2.2.3 RTL Output Functions

signalformat(arg)

```
reg reg1[7:0]@30 {
  signalformat("my_bitfield__%s");
  bit bit1@0;
  bit bit2@1 {
     signalformat("%s");
  }
}
```

Figure 8 - signalformat()

Most naming conventions prefix names with letters that define the signal. Signalformat() takes a string with a `%s` placeholder for the bitfield name. Since a key benefit of RDL is that the bitfield name in the specification matches the bitfield name in the RTL, the placeholder (`%s`) is required. Just like reset(), the signalformat() function can be specified at one level (in this case `reg1`) and then overwritten at another level (`bit2`). In Figure 8, `bit1` will be called `my_bitfield_bit1` but `bit2` will just be called `bit2`.

implement(arg)

In some cases, specific implementation details need to be provided. This is done using the implement() function. There are currently two options: implement(atomic) and implement(ram). Implement(atomic) is used to specify that the register must be accessed in its natural size. In the event that the register bus is narrower than the register being accessed, tanking will be implemented to read and write the register in its natural size. Tanking is a process where the register is atomically manipulated. On reads, the register is captured upon reading the lower half. When the upper half is then read, the captured upper half version is returned. On writes, when the lower half is written, it is stored but the register is unmodified. When the upper half is written both parts are written at the same time to the register.

Implement(ram) is used when an alternative solution for the registers will be used rather than having the RTL generator handle it. This is typically used for RAM models. In these cases, the RTL generators should not generate anything, and instead allow it to be added by the engineer.

3.0 reggen – an RDL parser/generator

Reggen is an internally-created tool that converts RDL input into various outputs, functioning in two phases.

3.1 Phase I – Parsing the language

The first phase is parsing. This phase is common to all RDL generators. Besides simple syntax checking, the source in evaluated to be valid registers. That is – tests such as register alignment and, overlapping address space are checked. In the process of doing this, the source file can be organized into a data structure to be parsed by output generators.

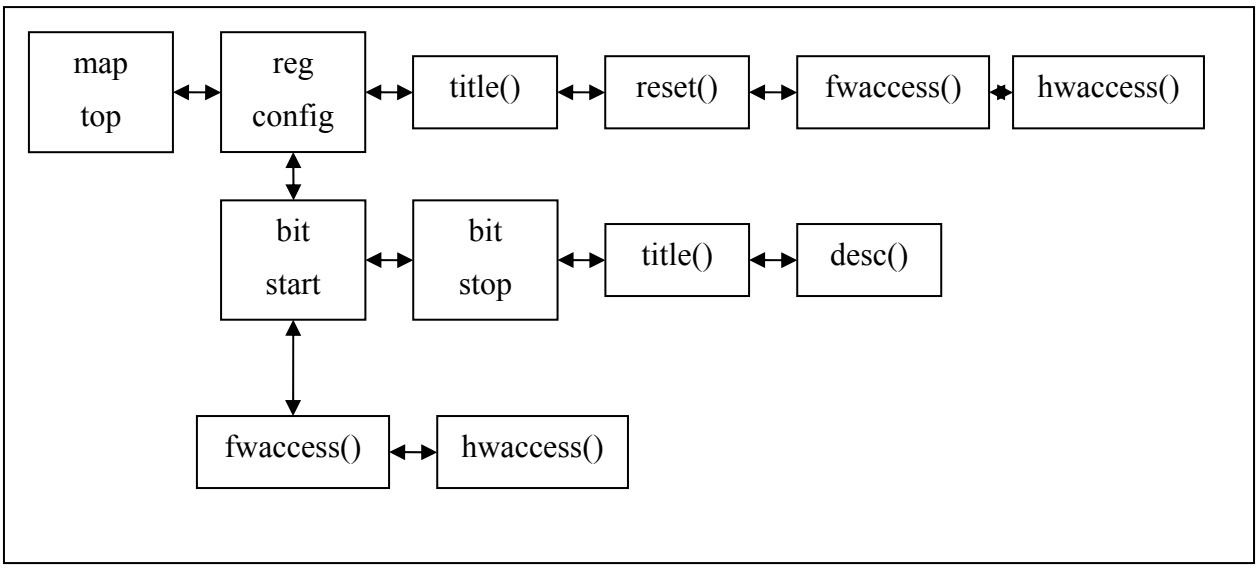


Figure 9 - Structure

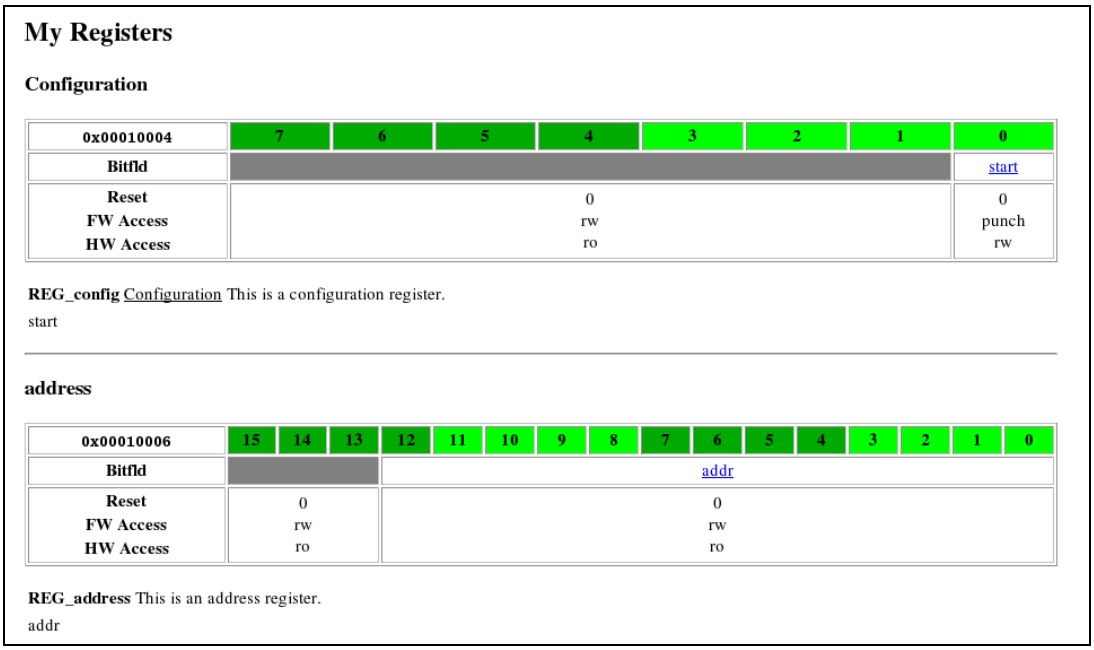
Figure 9 is a structural representation of Figure 4 showing how elements and functions are grouped together in doubly linked lists. Tools can then scan this structure to produce the proper output.

3.2 Phase II – Generating Output

Once the input has been validated, all generators should run without error. Generators can be grouped into three types: documentation, RTL, and software.

3.2.1 Documentation Generators

The documentation generators follow a common format. That is, they walk the list of registers and describe them and their bitfields. Two very useful formats are MIF and HTML. MIF format is the Frame Maker Interchange Format. It is a text-based format that can completely describe anything that can be placed in its native fm format. MIF is a very useful format for paper documentation.



My Registers

Configuration

0x00010004	7	6	5	4	3	2	1	0
Bitfld								start
Reset	0							0
FW Access	rw							punch
HW Access	ro							rw

REG_config Configuration This is a configuration register.

start

address

0x00010006	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bitfld				addr												
Reset	0			0												
FW Access	rw			rw												
HW Access	ro			ro												

REG_address This is an address register.

addr

Figure 10 - HTML Output

HTML format is an ideal online format. HTML is not limited to specific width and height limitations, and thus allows more freedom in representing registers. Figure 10 shows the html output of Figure 2.

3.2.2 RTL Generators

There are two sets of generators that are very useful for RTL: Bus Interface & Simple Flops. The Bus Interface block handles all write strobe generation and read muxes for the section. The Simple Flops block addresses a set of flops that are hwaccess(ro) and fwaccess(rw).

Bus Interface Block

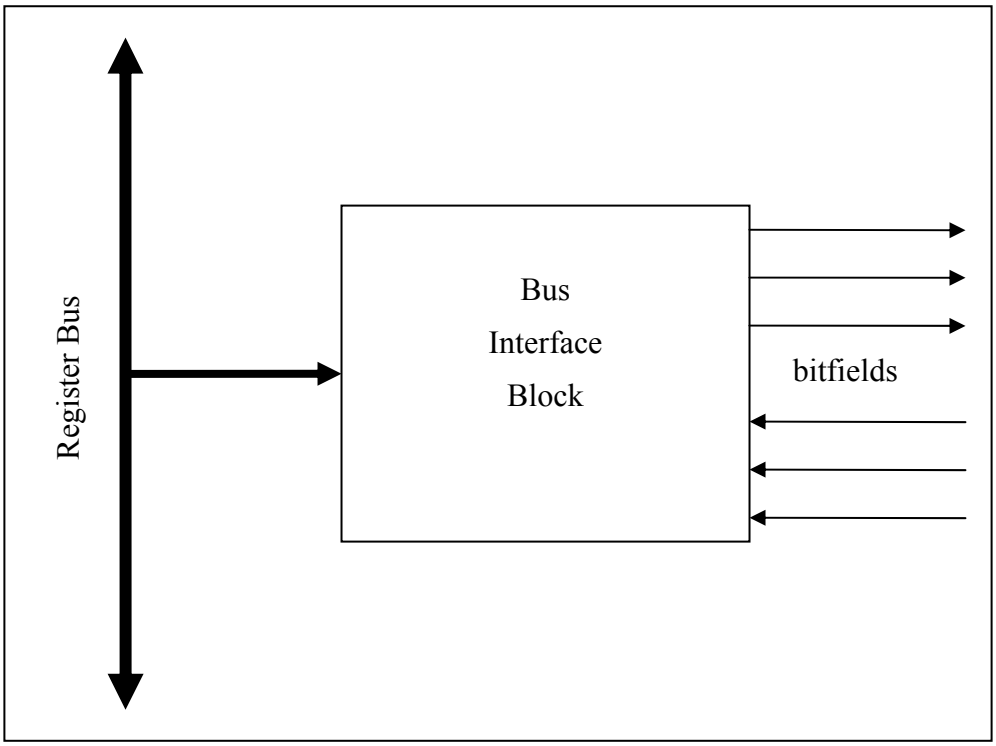


Figure 11 - Bus Interface Block

Register Buses change from design to design. Even registers themselves may change, with bitfields relocated from one register to another. The Bus Interface Block allows you to abstract that away. The RDL source is descriptive enough to know about everything in between. Issues such as tanking registers and dealing with bus widths are all well defined.

Simple Flops Block

Many bitfields are hardware read-only and firmware read/write. These are typically configuration registers and the sort. All that the hardware machines really need to see are wires that contain the state of these flops. The flops themselves are trivial. Also – all the information regarding these flops is well defined in the RDL source, and therefore can be easily generated.

3.2.3 Software Generators

In order to use your chip, you need to define registers in some form for firmware. This may be as complex as a C++ object, or as simple as some #defines. All these forms need information about registers such as their address, size, bitfields, etc, which is all available in the RDL source. Also, since the same RDL source generates the RTL, the bitfield names are common between hardware and firmware. This maintains consistent terminology between the engineers who design the chips and those who use the chips.

4.0 Issues with reggen, the RDL Language and future Enhancements

While RDL2.0 is very descriptive, it suffers from being too verbose. One example is in addressing. All elements need to have an address. This allows freedom of placement of the elements in the source file. However, it also means that you need to manually calculate the

addresses. The language could allow location-based addressing. That is, the order of the registers in the file is their address order. This would simplify entry.

Another example of the verbosity of RDL2.0 is when you have a 16-bit register with a 16-bit bitfield in it. RDL2.0 requires that you list both the register and the bitfield. A combined register/bitfield type might make sense in these cases.

RDL2.0 using Verilog range constructs (e.g. `[4:0]`) to specify sizes. While this might be useful, most implementations only care about the total size and not the range. It may make more sense to use the C/C++ method (e.g. `[5]`).

RDL2.0 does not support for macros. Often there could be similar yet not identical registers that need to be defined twice.

The reggen tool tightly links parsing and generation. This does not allow people to define new generators without recompiling the tool. Using some form of generator language or API could avoid this issue.

5.0 Acknowledgements

A key help to creation and improvement of any language is to have it tested. I would like to thank the Maxtor ASIC team for working though many of the issues necessary to make RDL2.0 a success. I would also like to thank Mark Johnson. Mark is the member of the SNUG Boston Technical Committee assigned to work with me on this paper.

6.0 More Information

Reggen and other information may be available upon request. Please contact me at julian_gorfajn@maxtor.com.