GEN_CSRS User Manual

Table of Contents:

1. Introduction
2. Installation
3. Usage
4. Example of input description
5. Format description
    5.1 General Options
    5.2 Register and Field Definition
    5.3 Correctness constraints
    5.4 Packet definition
6. Generated files
7. Command line help
8. Support
9. License

1. Introduction

Gen_csr is a utility to generate a synthesizable hardware description in verilog for a set of control/status registers based on a format textual description of their addresses, read/write attributes and fields. The registers are described on a formal XML format. The total set of optional outputs generated by the utility are:

- Synthesizable Verilog unit
- HTML documentation
- Header files to access the registers through macros from C/C++ or a verilog testbench
- RALF description for synopsys ralgen utility

2. Installation

This tool depends on some tools being preinstalled on the system:

- Emacs/xemacs being installed in the system and available on the path
- Perl
- Some perl modules that can get installed with the script provided (install_perl_modules)

To install the tool:

a) run 'install_perl_modules' script to ensure the perl packages required by the tool are installed in your system (run as root)

```
   $ sudo ./install_perl_modules
```

b) Run an example to check it is functional

```
        $ make
```

to generate the outputs from the sample file 'mycsrs.xml' provided

c) Inspect the generated file mycsrs.v and other generated files.

mycsrs.v shows interface to the host (AMBA APB) and implementation of several flavors of registers.

3. Usage

This utility generates a Configuration Status Register (CSR) block given a definition of the registers required written as an XML spec. The output is Verilog 95 with AUTO statements to be processed by Emacs/Xemacs Verilog mode auto expander (a copy of which is included for convenience but covered by a different license, see *.el header), or Verilog 2001 if -v2k option is passed

The register block allows communication from a Host side and from a hardware side or hardware Processing Element (PE). The PE side is optional and it is covered only in some specific sections of this document.

The XML specification of the register blocks defines:

- A set of registers, its internal fields and properties (R, RW, etc). A register has 3 aspects:

- The host side, how is it visible to the host. HOST_ATTR defines how the host can access this register.
- An *optional* PE side (not generated by default). A PE is a HW Processing Engine attached to the crs block. Basically this is the piece of HW that the csr block intends to control. How the registers can be accessed from the PE is defined by PE_ATTR. For instance a register can be RW from the host and read-only (R) from the PE. Additionally another register can be writable by the PE (W) and read-only from the host (R). To promote good design practices, all registers writable from host are also readable from it (I.e. no host side W-only allowed)
- A set of ports. Each register can generate few outputs to control other specific actions or can take values from the external world as inputs. These aspects are defined by the IO_ATTR of the register. For example a register can be defined as "in" type attached to a HW input port and can be defined a read-only (R) from host and PE sides. Or it could be defined as RW from the host, R from PE and additionally as an output port (out IO_ATTR)

The host accesses the CSR block through a single AMBA APB v 2.0 or v 3.0 bus, one register at a time.

The PE accesses all PE visible registers through explicit separate ports. All of them can be accessed simultaneously.

Additionally all "out" ports are available to external HW at any point of time and all "in" ports are subject to sampling at any point of time when either host or PE access the registers.

"in" and "out" ports can also generate an associated "trigger" signal if a TRIGGER_ATTR is defined. For instance if TRIGGER_ATTR="R" then *reading* the register by either host or PE generates a 1 clock cycle wide pulse available as an output from the csr block. Similarly if TRIGGER_ATTR="W" the output pulse will be generated when the value is written by the PE or host. With this feature external actions can be triggered upon reading/writing any register field.

The level of flexibility provided in the definition of the register fields also implies some of the combinations are not allowed and will be rejected by gen_csr. For example HOST_ATTR="R", PE_ATTR="R" and IO_ATTR="out" yields an error as nobody defines the value of the register.

Additionally, the file can be used to define convenience packets. Packets are abstractions that allow the source code to be consistent with the HW by defining macros and constants that abstract bit fields declarations, values, power-on reset values etc. For example one can define a packet of four words that are written into a FIFO where each word has different bit fields. Having a common definition allows the hardware that interprets this packet to be written in terms of Verilog macros automatically generated by this tool and allows the CPU to generate the packet format according to macros defined in C compatible header files. This makes the code more robust to changes (expanding width of a field or moving it to a different position in a packet word) could be done without modifying verilog or C code, and more robust to out of sync with spec bugs, as the spec/macros are both defined from a common source (XML description). Packet definitions generates no HW output.

4. Example of input description

Sample file: regs_test.xml

```
<REGISTERS>
   <DEFAULTS
       PRI="HOST"
       CLOCK="PCLK"
       PREFIX_CSRS="PE0_"
       PREFIX_PKTS="P_"
       GEN_MMIO="1"
   />
   <REG_ARRAY>
       <REG NAME="reg0" OFFS="0x00" MMIO="0">
              <FIELD NAME="n0" BIT_RANGE="0" HOST_ATTR="R" PE_ATTR="R" IO_ATTR="in" />
              <FIELD NAME="n1" BIT_RANGE="2:1" HOST_ATTR="RW" PE_ATTR="R"
                     IO_ATTR="internal" RESET="1"/>
              <FIELD NAME="m0" BIT_RANGE="9:8" />
       </REG>
```

```
        <REG NAME="reg1" OFFS="0x04" MMIO="1">
               <FIELD NAME="n2" BIT_RANGE="2:1" HOST_ATTR="RW" PE_ATTR="RW"
                      IO_ATTR="out_trigger" RESET="2"/>
               <FIELD NAME="n3" BIT_RANGE="5:3" HOST_ATTR="RC" CONST="33"
                      PE_ATTR="R" RESET="3"/>
        </REG>
        <REG NAME="status" OFFS="0x08">
               <FIELD NAME="f1" BIT_RANGE="2:1" HOST_ATTR="R" PE_ATTR="W"
                      IO_ATTR="out" RESET="2"/>
               <FIELD NAME="f2" BIT_RANGE="9:8" HOST_ATTR="R" PE_ATTR="W" RESET="3"/>
        </REG>
        <REG NAME="control">
               <FIELD NAME="x1" BIT_RANGE="2:1" HOST_ATTR="RW" PE_ATTR="R"
                      IO_ATTR="in_trigger" RESET="2">
                      <ENUM_LIST>
                             <ENUM NAME="IDLE" VALUE="1" />
                             <ENUM NAME="NEXT" />
                             <ENUM NAME="END" />
                      </ENUM_LIST>
               </FIELD>
               <FIELD NAME="x2" BIT_RANGE="31:3" HOST_ATTR="RW" PE_ATTR="R" RESET="0x20"/>
        </REG>
        <REG NAME="intr_status">
               <FIELD NAME="type0" BIT_RANGE="0" HOST_ATTR="RW1C" PE_ATTR="W" />
               <FIELD NAME="type1" BIT_RANGE="1" HOST_ATTR="RW1C" PE_ATTR="W" />
               <FIELD NAME="regular" BIT_RANGE="9:8" />
        </REG>
    </REG_ARRAY>
    <PKT_ARRAY>
        <PKT NAME="pkt0">
               <FIELD NAME="q0" BIT_RANGE="0" />
               <FIELD NAME="q1" BIT_RANGE="2:1" />
               <FIELD NAME="q2" BIT_RANGE="9:8">
               <ENUM_LIST>
                      <ENUM NAME="IDLE" VALUE="1" />
                      <ENUM NAME="NEXT" />
                      <ENUM NAME="END" />
               </ENUM_LIST>
               </FIELD>
        </PKT>
        <PKT NAME="pkt1">
               <FIELD NAME="p0" BIT_RANGE="0" />
               <FIELD NAME="p1" BIT_RANGE="2:1" />
               <FIELD NAME="p2" BIT_RANGE="9:8" />
        </PKT>
    </PKT_ARRAY>
</REGISTERS>
```

To process it do:

```
$ gen_csrs regs_test.xml
```

5. Format description

5.1 General Options

The following options are defined under the "DEFAULTS" sections of the XML input file. For example:

```
<REGISTERS>
   <DEFAULTS
       CLOCK="PCLK"
       PREFIX_CSRS="GCSR_"
       PREFIX_PKTS="P_"
       PREFIX_PORTS="1"
       APB_ADDR_W="20"
       BASE="0xC0000"
       DOC="PE1 register definition"
   />

   ... other sections follow ...

</REGISTERS>
```

The allowed **DEFAULTS** fields are the following:

DEFAULTS attributes
CLOCK : defines the clock name of the generated verilog module Legal values: any verilog-and-c valid identifier Default: *PCLK* **CLOCK_EDGE** : defines the clock active edge using verilog notation Legal values: posedge|negedge Default: *posedge* **RESET** : defines the name of the reset signal

Legal values: any verilog-and-c valid identifier
Default : *PRESETN*

RESET_VAL : value of the reset line that puts the system IN reset
(use 0 for an active-low reset, 1 for an active-high one)
The reset generated is asynchronous

Legal values: 0|1
Default: 0

BASE : defines the value added to all register offsets to complete the byte address of each register

Legal values: a 32 bit value in decimal or c-style hex format (e.g. 256 or 0x100)
Default: 0

OFFS_INCR : automatic offset increment to be applied if offset not specified for a register

Legal values: a 32 bit value in decimal or c-style hex format
Default : 4

APB_ADDR_W : width of the APB address bus. Must be sufficient given the register addressed defined in the file

Note1: APB address will be declared as input [APB_ADDR_W -1 : 0] PADDR;

Legal values: a 32 bit value in decimal or c-style hex format
Default : 32

PREFIX_CSRS : string prepended to the name of all registers. Useful if the user wants to prepend a module name in front of all the registers and at the same time make the register definitions less verbose

Legal values: any verilog-and-c valid identifier
Default: ""

PREFIX_PKTS : string prepended to the name of all packets defined

Legal values: any verilog-and-c valid identifier
Default: ""

PREFIX_PORTS: field names become module ports in several cases depending on register IO attributes. For those that become ports this attribute defines whether the register name prefixes the field name or not to form the full port name. If not the field name must be unique among all registers to avoid name conflicts.

Legal values: 0|1
Default: 1

APB_V3 : if 1, an APB version 3.0 is generated, if 0 a version 2.0 one

Note2: If APB 3.0 interface is chosen the block will drive PREADY and PSLVERR signals out

Legal values:	0|1
Default:	0

GEN_MMIO : if 1 it generates logic and ports to allow the registers to be written from a Hardware Side/port. This port is typically another processing element called PE in this document

Legal values:	0|1
Default:	0

5.2 Register and Field Definition

Example:

```
<REGISTERS>
   <DEFAULTS
   ... see previous section ...
   />
   <REG_ARRAY>
       <REG NAME="reg0" OFFS="0x00" MMIO="0">
       <FIELD NAME="n0" BIT_RANGE="0" HOST_ATTR="R" PE_ATTR="R" IO_ATTR="in" />
       <FIELD NAME="n1" BIT_RANGE="2:1" HOST_ATTR="RW" PE_ATTR="R"
                      IO_ATTR="internal" RESET="1"/>
       <FIELD NAME="m0" BIT_RANGE="9:8" />
       </REG>
       <REG NAME="reg1" OFFS="0x04" MMIO="1">
       ...
       </REG>
   ...
   </REG_ARRAY>
   ...
</REGISTERS>
```

Each **REGISTER** can have the following attributes:

REGISTER attributes

DOC: A documentation string used as a header for the auto-generated documentation. DOC field is reported in the summary for every register and in the documentation for it.

Legal values:	any string
Default:	""

DOCEXT: Extended documentation. Extra explanation for the register functionality. It is reported in the register documentation but NOT in the summary table for the registers

Legal values:	any string
Default:	""

DOC_ATTR: If declared as private, the register fields is not documented in the HTML output.

Legal values:	private|public
Default:	public

NAME: the name of the *register*. Must be unique

Legal values:	any verilog-and-c valid indentifier
Default:	none - this field is mandatory

OFFS: offset to be added to BASE to yield the final register address from the host side

Legal values:	a 32 bit value in decimal or c-style hex format
Default: 0.	Incremented by OFFS_INCR on each new register definition

COUNT: Can be used to generate a set of arrayed registers with the same layout. They allow some nice features over just replicating the register definition with incremental names that could be performed with a pre-processor for instance.

- More compact generated HTML documentation
- Indexed access macros on generated header files.

The registers get an automatically generated incrementing suffix starting at **IDX0**

Legal values:	1..
Default: 1.	Incremented by INCR on each new register definition

NOTE: Please see scrtach_pad_a /scratch_pad_b / scratch_pad_c / scratch_pad_d registers in mycsrs.xml for an example of use.

IDX0: The index number of the first register in an arrayed set. This attribute is only relevant if COUNT attribute is greater than 1

Legal values:	0..
Default: 0.	Incremented by 1 on each new register definition

INCR: Incrementing address step for arrayed registers

Legal values:	1..
Default: 4.	Address is incremented by 4 by default

MMIO: If GEN_MMIO=1 then this field defines the MMIO port/channel where the register will be visible to the PE

Legal values:	a 32 bit value in decimal or c-style hex format
Default:	0. It will be incremented by 1 on each new register definition

Additionally *a register must have one or more* ***FIELDS***.
Each FIELD may contain the following attributes

FIELD attributes

NAME: register field name. Must be unique within the register. If PREFIX_PORTS=0 then it must also be unique among all field names in the same XML file (or csr block)

Legal values:	any verilog-and-c valid identifier
Default:	none - this field is mandatory

BIT_RANGE: msb[:lsb]

Legal values:	Both msb and lsb must be integers in the range 0 through 31 with msb >= lsb. If lsb is omitted it will be assumed lsb = msb
Default:	none - this field is mandatory

HOST_ATTR: defines the register properties from the APB host side

Legal values:

RW:

The register can be Read and Written from host side

RW1C:

The register can be Read from host side and Writting it affects the register in the following way:

- Bits written as 1 clear the corresponding bit in the register
- Bits written as 0 do not modify the corresponding bit in the register

(this mode is typically used in interrupt status registers)

RC :

Read constant value. The constant value returned is defined by the CONST attribute if provided (0 is the default if CONST not provided)

R :

The value is read-only for the host

Default: RW

IO_ATTR: defines the I/O attributes of the register, or in other words whether the register is tied-up to a input or output port and in which way

Legal values:

in :

The register field is declared as an input to the csr block

out:

The register field is declared as an output from the csr block. Any field can be defined as output regardless of the value of HOST_ATTR and PE_ATTR

internal:

The register field does not generate any input/output port on the csr block

lintr:

Interrupt to be latched. Assumed active high. It generates

- an input field called <field_name>_intr. This is the external interrupt line coming in to be latched
- An output field called <field_name> with corresponding internal FF's. The internal logic latches the transition to high value of the bits of the field. It is currently required that the host or pe has write access to the field (RW or RW1C) to allow a path to clear this field.

Default: internal

TRIGGER_ATTR: defines whether accessing the register generates a side-band signal that can be used by the hardware

Legal values:

R :

An output port called <reg_name>__<port_name>_trigger_rd_D is generated. It is pulsed by one clk on any host or PE read the register (the register can be internal or in type)

W:

An output port called <reg_name>__<port_name>_trigger_wr_D is generated; it is pulsed by one clk on any host or PE write the register (the register can be internal or out type). Additionally a bus called <reg_name>__<port_name>_D is also generated containing the new value to load into the register as a consequence of this write.

RW:

All the signals above are generated and triggered for reads or writes respective

NONE:

No special trigger signals are generated

Default: NONE

RESET: defines the default value taken by the register field upon reset assertion

Legal values: a value in decimal or c-style hex format within the size of the register field

Default: 0

DOC: A documentation string used as a header for the auto-generated documentation. DOC field is reported as a description of the register field.

Legal values:	any string
Default:	""

PE_ATTR: defines the attributes from the HW controlled by the csr block (processing element or PE in this document)

Legal values:

RW :	The register can be Read and Written from the PE
W :	The register can be Written from the PE
R :	The register can be Read from the PE

Default: RW

5.3 Correctness constraints

HOST_ATTR = R and PE_ATTR = R must have IO_ATTR = in
HOST_ATTR = RC and IO_ATTR = in|lintr is invalid
Register fields must have non-overlapping bit ranges
Field reset and constant values must fit in the number of bits allocated to the field

More details below

Usual cases for no PE port (GEN_MMIO="0", i.e. pure host access)

Case	HOST _ATTR	IO _ATTR	TRIGGER _ATTR	FF generated	Port	Usage
1	RW	Internal	NONE	Y	n/a	Scratch-pad register
2	RW	in	NONE	N	I	Status register externally implemented. Writes are discarded. Candidate for INVALID. Should use R only
3	RW	in	R/W/RW	N	I	Register externally implemented, host R/W notify the external HW block
4	RW	out	NONE	Y	O	Control register going out of CSR block
5	RW	out	R/W/RW	Y	O	Control register going out of CSR block. Host Read or Write triggers external action
6	RW	Iintr	R/W/RW	Y	IO	Input is internally latched. The input generated as <field_name>_intr. The latched version is output with no suffix as <field_name> Trigger signal is generated on access. Not a very common case
7	RW	Iintr	NONE	Y	IO	Input is internally latched. The input generated as <field_name>_intr. The latched version is output with no suffix as <field_name>
8	R	Internal	NONE	*	-	Will always read back reset value (Not very useful)
9	R	in	NONE	N	I	Status register externally implemented
10	R	in	R	N	I	Register externally implemented, host Read will notiify the external HW block (may be for clear on read functionality or if reading a FIFO to trigger the pop from it)
11	R	in	RW/W	-	-	INVALID combination
12	R	out	NONE	Y	O	Will always read back reset value and will drive it out (Not very useful)
13	R	out	R	Y	O	Will always read back reset value and will drive it out (Not very useful). Host read triggers external action

14	R	out	W/RW	-	-	INVALID combination
15	R	Iintr	-	-	-	INVALID combination
16	RC	Internal	NONE	N	n/a	Will always read back a constant value (e.g. an ID or version number)
17	RC	in	-	-	-	INVALID combination
18	RC	out	NONE	N	O	Will always read back constant value and will also drive it out as control value
19	RC	out	R	N	O	Will always read back constant value and will also drive it out as control value Host read triggers external action
20	RC	out	W/RW	-	-	INVALID combination
21	RC	Iintr	-	-	-	INVALID combination
22	RW1C	Internal	NONE	Y	n/a	Register whose bits can only be cleared (until a later reset)
23	RW1C	in	NONE	N	I	Status register externally implemented (?)
24	RW1C	in	R/W/RW	N	I	Register externally implemented, host R/W notify the external HW block
25	RW1C	out		Y	O	Register whose bits can only be cleared and value goes out. Can be used for bits that enable some time of operation and once locked (set to 0) can not be turned back until a block reset.
26	RW1C	out	RW	Y	O	Register whose bits can only be cleared and value goes out. Can be used for bits that enable some time of operation and once locked (set to 0) can not be turned back until a block reset. Access triggers something in the HW
27	RW1C	Iintr	NONE	Y	IO	Input is internally latched. The input generated as <field_name>_intr. The latched version is output with no suffix as <field_name> This is the most common way to define an interrupt signal internally latched

28	RW1C	lintr	R/W/RW	Y	IO	As 27 but with associated trigger signals on access. Not clear what is the use

* whether a FF is implemented or not here depends on synthesis tool (most won't implement a FF)

Case 3 is very special and can be used as a catch-all option when nothing else fits adequately. It allows using this generator to take care of the APB interface and documentation while leaving the register completely implemented OUTSIDE the block and fed through an input port. An external register can be implemented as follows.

```
`include "csr_host_defines.vh"

reg [W-1:0] ext_field;
always @(posedge PCLK or negedge PRESETN)
   if (~PRESETN)
       ext_field <= `CSR_FLD_RST_<RegName__FieldName>;    // note that XML reset value is
                                                          // used to generate this constant
   else if (....)
       ...                                                // your custom logic here
   else if (<RegName__FieldName>_wr_trigger_D)
       ext_field <= <RegName__FieldName>_D;
end
```

Note that the trigger signals are a single bit, regardless of the width of the associated register field.

ext_field should be fed back as an input with the name of <RegName__FieldName> so that the read logic gets the externally implemented value on the host/PE.

Note that the value of <RegName__FieldName>_D used to write the new value of the field already incorporates byte enable logic (only bits aligned with enabled bytes take the new value) and whether the bits are RW1C.

'lintr' type of ports are used to generate a status bit for an line that acts as an high-true interrupt. It usually involves sanitizing the signal to become a single clock pulse signal, latch it and provide a mechanism to clear the interrupt from the host or PE. If a <field_name> is defined as IO_ATTR="lintr" then:

- <field_name>_intr is generated as an input. The user should connect it to the input edge interrupt to this port
- <field_name> is generated as an output. It contains the latched interrupt and can be used a CPU level interrupt line. The state of this latched interrupt becomes available to the host/PE on an internal field of the same name for read-back or clearing.

5.4 Packet definition

Packets definitions generate macros and c/verilog defines that can be used to rise
the level of abstraction of the code (both verilog and c/c++) and make it more consistent with each other without having to hardcode bit-field ranges for specific purposes. Packets don't generate hardware, but they help on defining data-structures with specific bit layout in a portable way.

This is an example of packet definition:

```
<REGISTERS>
   <DEFAULTS
   ... see previous sections ...
   />
   <REG_ARRAY>
   ... see previous section ...
   </REG_ARRAY>
   <PKT_ARRAY>
       <PKT NAME="pkt0">
       <FIELD NAME="q0" BIT_RANGE="0" />
       <FIELD NAME="q1" BIT_RANGE="2:1" />
       <FIELD NAME="q2" BIT_RANGE="9:8">
       <ENUM_LIST>
              <ENUM NAME="IDLE" VALUE="1" />
              <ENUM NAME="NEXT" />
              <ENUM NAME="END" />
       </ENUM_LIST>
       </FIELD>
       </PKT>
       <PKT NAME="pkt1">
       ...
       </PKT>
       ...
   </PKT_ARRAY>
</REGISTERS>
```

The format is a simplified version of the register definition format. Only packet/field
name and bit ranges are relevant.

6. Generated files

This section lists the default files generated. The names of all of them can be overridden with command line parameters. See gen_csr -help for syntax details.

- **csr_pe_defines.h:**

Generates constants and access macros for each register and field. The are intended to be used on the code running on the PE

- **csr_host_defines.h:**

Generates constants and access macros for each register and field. The are intended to be used on the code running on the HOST. The main difference between the two files above is the register address

- From the host point of view the address is defined by BASE+OFFS attributes
- From the PE point of view the address is a number starting at 0 and incremented by 1 by default for each new register.
  The correspond to MMIO channels 0..n or given by the MMIO attribute

- **csr.html**

HTML documentation for the register block

- **regs.v.plp**

actual csr block verilog file. Needs to be postprocess by plp

```
$ plp regs.v.plp
```

- **regs.pm**

A Perl data structure result of parsing the input XML file. It can be included in any Perl script doing:

```
use regs;
```

to get access to all the fields defined in the input XML file

- **pkt_defines.h**

Definitions and macros to access the fields of the defined packets

- **csr_mmios.h**

Declares as set of classes that can be used on the PE side (assuming the PE is a CPU type of processing engine) to read/write the registers. For example

```
#ifndef CSR_MMIOS_H_INCLUDED
#define CSR_MMIOS_H_INCLUDED
```

```
CSR<0> reg0 ;
CSR<1> reg1 ;
SR<2> status ;
CR<3> control ;
CSR<4> intr_status ;

#endif
```

7. Command line help

For the most up-to date information:

```
$ gen_csr -help
```

```
gen_csrs Rev 1.48. Generation of Control/Status verilog register file and documentation based on XML description

   Usage: ./gen_csrs [options] xml_filename

   Where options is any combination of the following (can be abbreviated to unique)

      -out f              filename of the output register module (default regs.v)
      -mod s              define module name as <mod>
      -mmio_defines f     filename of the defines and macros for mmio usage (default <mod>_mmios.h)
      -host_defines f     filename of the defines and macros for host usage (default <mod>_host_defines.h)
      -host_vdefines f    filename of the defines and macros for host usage in verilog
                          (default <mod>_host_defines.vh)
      -pe_defines f       filename of the defines and macros for PE usage (default <mod>_pe_defines.h)
      -pkt_defines f      filename of the defines and macros for pkt processing (default _pkt_defines.h)
      -csr_params f       filename of the csr parameters file (not generated unless provided)
      -ralf f             filename of the output ralf file generated (not generated unless provided)
      -html f             filename for the output HTML documentation (default <mod>.html)
      -[no]parallel_mode  in Parallel_mode PREADY/PRDATA's from several gen_csr modules can be OR'ed together and
                          PSLVERR is never asserted.
                          (default 0)
      -[no]gen_regs       generate register portion of the output (default 1)
      -[no]gen_pkts       generate Command portion of the output (default 1)
      -[no]hyperlinks     Create hyperlinks from summary to each register(default 0)
      -[no]warn_aw        Warn if constants specified are larger than expected size (default 1)
      -vlogsep s          Separator for register name and field in verilog ports (default __)
      -sep s              Separator for register name and field for everything else (default __)
      -cq n               Include this delay on non-blocking assignments to help waveform debugging. None if 0 (default 0)
      -[no]v2k            Use verilog 2000 syntax (default 1)
      -[no]pstrb          Include PSTRB input control (default 0)
      -rdata_on_decerr vn Value returned on read if reading a non-existing/non-readable register
                          (default 32'hbad00add)
      -[no]debug          Enable debug messages (default 0)
      -help               Display this message

  Where: f is a valid filename (E.g /path/to/out.txt)
         s is a string then conforms to C identifier rules (E.g. value_max)
         n is a natural number  (E.g 2)
         vn is a verilog formated number/constant (E.g. 32'hdeadbeef)

         <mod> is extracted as the basename of the xml_filename if no provided (e.g. /path/timer_csrs.xml
         yields timer_csrs)
```

8. Support

Please e-mail Miguel Guerrero <miguel@esenciatech.com>

9. License