

csrGen: Automated CSRs for ASIC/FPGA Processor Interfaces

Chuck Benz

Chuck Benz ASIC and FPGA Design

cbenz@chuckbenz.com

ABSTRACT

csrGen is a tool to automatically produce synthesizable verilog RTL code for the registers that make up the memory map of a processor interface from a simple template that lists and describes the registers. Using a tool for registers speeds ASIC or FPGA development, avoids common errors, and can also aid documentation, verification, and firmware development.

Table of Contents

Table of Contents	2
Table of Figures	2
1 Introduction	3
2 CSR Operation and Types of CSRs	3
3 Implementing CSRs in an ASIC/FPGA Design	4
3.1 Implementation Errors	5
3.2 Automating CSRs with an EDA Tool	5
4 The csrGen Template File Syntax	6
4.1 A Simple Example	6
4.2 csrGen Structure Declarations	8
4.3 Address Declarations	8
4.4 csrGen CSR Properties	9
4.5 csrGen Module Properties	10
4.6 Flop Properties	11
4.7 Direct Verilog Code	11
5 Design Verification	12
6 Future Enhancements and Improvements	12
6.1 csrGen Enhancements	12
6.2 Other Directions	13
7 Conclusions and Recommendations	13
8 Acknowledgements	13
9 References	13

Table of Figures

Figure 1 Example csrGen Template File	6
Figure 2 Example csrGen verilog output	7

1 Introduction

Many ASIC and FPGA designs include processor interfaces. Beyond the logic specific to how the particular processor bus operates, these interfaces consist mainly of a set of control/status registers (CSRs), described by a memory map spanning many addresses. Often these register sets are quite large, and there may be a variety of behaviors associated with different fields in the registers.

The logic structures to implement these CSRs are repetitive and relatively simple. This makes automating the task of RTL coding them attractive. Automating the task has several advantages:

- Time savings – the RTL coding is verbose, and different code fragments relating to one CSR field are scattered in many different places in the RTL: declaration, module I/O list, sensitivity list, read logic, write logic, reset and clock logic. Automation allows one simple specification of a field to propagate to all of those code fragments.
- Correctness – with all of those code fragments for one field dispersed, errors are possible.
- Consistency – an automated tool tends to suggest preferred structures (not by limiting options, but by making the preferred way easy), and simplifies the use of templates, examples, or descriptions of structures to be used for all chips in a project.
- Single source code – a specification that is more than the source for RTL, also generates:
 - the verification environment, including automated testing of CSR fields,
 - embedded firmware, which usually needs a .h file to describe field locations,
 - documentation.

This paper describes `csrGen` which is a PERL script to automate the CSRs in a processor interface. It was written with these goals in mind:

1. Simple but powerful template format to describe CSR fields.
2. Quick operation to produce updated RTL after any change.
3. Many different types of CSR field behavior.
4. Allow for functionality beyond register read/write access (integrate user written RTL directly in the template).
5. Create synthesis-ready Verilog RTL code (`csrGen` could be adapted for VHDL).
6. Create CSR field definitions for C, Verilog, and Vera.
7. Create code fragments for the module that instantiates the processor interface (wire declarations and an instantiation).

`csrGen` is freely available at <http://asics.chuckbenz.com>

2 CSR Operation and Types of CSRs

A basic CSR is an "entity" that a processor can read and/or write at a specific address and bit position(s). This implies a set of flops, a write multiplexer controlled by a write strobe and an address decode, and a read multiplexer controlled by an address decode. This can be termed a "read/write" CSR.

Many applications require a broad variety of different CSR properties, often combining several of these properties. A small subset of common CSR types is summarized below as examples.

Read-Only: a common variation, with no logic for writing. Sometimes this may be a constant value, for example a chip ID or version number.

Clear-on-read: the read operation returns the value of the CSR, but then the contents are reset to zero. A similar property would be that the value is reset to zero by writing a non-zero value – the intent being that writes can be made to other fields (usually control fields) at the same address without modifying this field (usually a status or counter field). For multi-bit fields, some implementations of this might reset all bits if any non-zero value is written, others might do a bit-by-bit write-1-to-clear and require all 1's to clear the multi-bit field.

'Sticky': often used to describe register bits that are set and remain set in response to an event. Often these may be events that propagate as an interrupt. The sticky bit would usually be cleared by a clear-on-read or write-1-to-clear operation.

Counters: another common CSR type. They also may be clear-on-read or write-1-to-clear.

Execute action: some CSR fields act as triggers – when written, some operation should be performed. The value written may or may not be significant. Sometimes the field retains its value until the operation is complete and can be read to determine completion status, or the field might not be read at all (write-only).

There are many other possibilities for CSR behaviors; these are simply the most common. Most designers and programmers first become familiar with basic read/write registers as part of a standard processor model, and then may learn about the variations as microprocessor peripherals or peripherals integrated in a microcontroller.

3 Implementing CSRs in an ASIC/FPGA Design

Most processor interface logic is coded in RTL by a designer, with some logic specific to how the processor bus operates, flops for the CSRs, read logic, write logic, and ancillary logic for special functions such as triggered operations, counters, and interrupts.

Sometimes this may be in a single module, or it may be spread over several modules in a design with one module for the actual chip IO. When the CSR logic is in a single module, most CSR fields are outputs from that module and connect to other modules that use the values. Or for status fields, they are inputs to the CSR module coming from other modules. Additionally, a file is created with definitions of each CSR's address, with and position in a general form that is easily transformed into C, verilog, or Vera define statements.

3.1 Implementation Errors

The CSRs can be an error prone area of a design for several reasons:

- The behavior is spread across many distinct locations of the RTL code, including: the module IO list and declarations, the flop reset value, the read logic, the write logic, and logic for any special properties.
- The logic is simple, so often designers are less careful in coding it, particularly when making a late change to a design. Often the focus is on the application of the CSR field that is being added, rather than the read/write access logic for it.
- Verification sometimes may focus more on control of chip operation (writes) rather than chip status (reads).

As a designer, I have often made late changes to CSRs in a design and stumbled over errors in sensitivity lists, module IO lists, and other minor syntax errors. I have also observed a bug in a released commercial IP core from a good vendor in which two bit positions were reversed between read and write operations.

3.2 Automating CSRs with an EDA Tool

Portions of the processor interface are obviously automated, but doing just those portions then requires a way to integrate the automatic parts with the rest. And the rest of the logic will differ for (almost) every design. One possibility is generating code fragments that are then assembled by a designer using a favorite text editor, but that requires repeating that step with every change (and change seems inevitable). Another might be using constructs like verilog's `include` construct, but many (including this author) reject that, preferring that the full RTL code be apparent when looking at the main file (and there may be file management issues).

A more elegant model of tool operation is the "automatics" present in verilog emacs mode [1]. This uses special comments which are expanded by the emacs editor to add the automatic code.

Automating RTL code for CSRs has been done by designers on many projects. `csrGen` is the second such tool I have developed. The first tool read a list of CSR definitions from a file (a custom syntax) and generated verilog code fragments and documentation fragments. Repeating the assembly of the fragments for each change became tiresome.

For `csrGen`, the decision was made to use a template file (syntax specific for `csrGen`) that would define CSR fields *and also* include as much verilog code as a designer might wish to integrate with the CSR code. The `csrGen` program processes the template and produces a verilog file (ready for synthesis) with the complete module. It also generates a file with verilog `wire` declarations for all module IO signals, and a file with a verilog instantiation – these files can be inserted into the verilog file for the next higher module in the hierarchy.

4 The csrGen Template File Syntax

4.1 A Simple Example

Figure 1 is a very simple example of a csrGen template file, and Figure 2 is the verilog RTL that is produced by csrGen.

```
%A 0
7:0 field1

%A 1
7:0 version RO

%A 2
3:0 field2
6  someerror sticky W1C

%I read
%I write
%I address 4
%I up_datain 8
%OF up_dataout 8

%VCL
if (write) case (address)
%writecase
endcase

if (read) case (address)
%readcase
endcase
%E

%AUTO
```

Figure 1 Example csrGen Template File

`%A` describes an address – the lines following `%A` define fields at that address, with optional properties.

`%I` describes an input, with an optional width.

`%OF` describes an output from flops, with optional width.

`%VCL` starts the user defined combinational logic, which ends with `%E`. The `%writecase` and `%readcase` are expanded by csrGen with all necessary write and read logic.

`%AUTO` selects the use of 'automatics' from emacs verilog mode for the module IO list and for the sensitivity list. csrGen can generate these directly, but this option may be preferred by designers already using emacs verilog mode (and it also cuts the number of lines in the output for our example, so it fits on just one page).

(This example uses just a small subset of the options and properties supported by csrGen).

```

module chip_up_ifc (/*AUTOARG*/) ;
input clock, init1 ;
input  [7:0]  version;
input      someerror;
input      read;
input      write;
input  [3:0]  address;
input  [7:0]  up_datain;
output [7:0]  field1;
output [3:0]  field2;
output [7:0]  up_dataout;
reg  [7:0]  field1, field1_D;
reg  [3:0]  field2, field2_D;
reg      someerrorS, someerrorS_D;
reg  [7:0]  up_dataout, up_dataout_D ;

always @ (/*AUTONSENSE*/) begin
    field1_D = field1 ;
    field2_D = field2 ;
    someerrorS_D = someerrorS | someerror ;
    up_dataout_D = up_dataout ;

    if (write) case (address)
0: begin
    field1_D = up_datain[7:0] ;
end
1: begin
end
2: begin
    field2_D = up_datain[3:0] ;
    someerrorS_D = (someerrorS_D & ~up_datain[6]) | someerror ;
end
    endcase

    if (read) case (address)
0: begin
    up_dataout_D[7:0] = field1 ;
end
1: begin
    up_dataout_D[7:0] = version ;
end
2: begin
    up_dataout_D[3:0] = field2 ;
    up_dataout_D[6] = someerrorS ;
end
    endcase
end

always @ (posedge clock or negedge init1)
    if ( ! init1) begin
        field1 <= 0 ;
        field2 <= 0 ;
        someerror <= 0 ;
        up_dataout <= 0 ;
    end
    else begin
        field1 <= field1_D ;
        field2 <= field2_D ;
        someerrorS <= someerrorS_D ;
        up_dataout <= up_dataout_D ;
    end
end
endmodule

```

Figure 2 Example csrGen verilog output

Highlighted in figure 2 is the verilog code that was written by the designer in the template file. In this example, very little was written by the designer, but it should be clear from the example that the designer can insert any desired statements to alter values before or after the case statements for write and read logic.

The coding style separates all combinational logic from the code for flops – this allows for the designer to add statements to modify the '_D' variables as desired with statements.

The generated code has the main structure shown:

- Comments from template file
- Module, input, output, reg, wire declarations
- Designer code from %v block in template file
- `always` verilog block for combinational logic
 - defaults for all _D inputs to flops
 - designer code from %VCL block in template file, with read and write cases expanded
- `always` verilog block for flops (clock and reset)

4.2 csrGen Structure Declarations

Aside from CSRs, csrGen provides for declaring and creating the usual variety of structures used in RTL coding, such as inputs, outputs, flops, and wires. The declaration format is somewhat simpler, so syntax errors are less likely. These declarations include:

- %I – input, with an optional width.
- %O – output, with an optional width.
- %W – wire declaration, with an optional width.
- %R – verilog `reg` declaration, with an optional width (creates no other structure).
- %F – flop, with an optional width, reset value, and direct value:
 %F <flopname> [width [resetvalue [directvalue]]]
- %OF – output from a flop, with an optional width, reset value, and direct value.

All of these also can be specified as repeated structures: %IREPEAT, %OREPEAT, etc..., with a repeat count specified after the name:

```
%FREPEAT <flopname> <repeatcount> [width [resetvalue [directvalue]]]
```

If the name contains a '%' character, it will be replaced by an index incrementing from 0 as the structure is repeated, otherwise the index will be appended to the name. A '%' character in the direct value will also be replaced by the index.

4.3 Address Declarations

The %A declaration begins definition of a set of CSRs at a specified address. Each following line is taken to define a CSR, and is expected to start with a numeric character. The next line that

begins with '%' is understood as the end of the CSRs for that address. The %A declaration format is:

```
%A <address> ["name or comment"] [properties] [writetaskname] [readtaskname]
```

The numeric address may be decimal or hex (in the form "0x1a"). Properties and any text in quotes may precede or follow the tasks. The first token not quoted and not recognized as a property will be taken as a verilog task name to be called when the address is written. The second such token will be taken as a verilog task name to be called when the address is read. The properties are a subset of the properties described below for individual CSRs, and apply to all fields at the specific address. The subset is: RO, COR, WIC (all defined below as CSR properties).

The calls to verilog tasks for write and read operations allow for the designer to add special logic that might be associated with triggering related operations. ("Execute action" as described in section 2).

A repeated form of %A is also supported as %AREPEAT with a repeat count following the address, which is the starting address for the repetition. If no further parameter is present, then the CSRs will be repeated at addresses incrementing by 1; otherwise the third parameter is the address increment between repetitions. If any of the optional parts of the %A declaration are desired, they must be preceded by an increment value, even if it is 1. If a CSR name in a %AREPEAT declaration includes a '%' character, it will be replaced by the incrementing index. If the CSR has the 'buss' property then the index is used as a bit subscript in the form name[index] (only for single bit fields), otherwise the index is appended to the name.

4.4 csrGen CSR Properties

This section lists the properties that may be set on individual register fields. Most can be applied in combination with others. These keywords can be in any order on the same line with the register field definition. Upper vs. lower case is not significant.

- Intern: internal – indicates that the field is not an output or input of this verilog module.
- RO: read only – the field is only present in the read logic, and is an input to the module (unless 'Intern' property is also declared).
- COR: clear on read – the field is reset to 0 when the address is read.
- WIC: write-1-to-clear – the field is reset to 0 when the address is written and the corresponding bit(s) is/are 1, but is not altered if the bit(s) is/are 0. (csrGen implements this on a per-bit basis for multi-bit fields).
- ST: sticky – the field is sticky. The name in the definition is taken as an input to the module (unless 'Intern'), and a flop is created for the sticky memory. The sticky value will remain 1 if the nominal value is ever 1, if even only for one clock cycle. In the example above, someerror becomes an input, someerrorS is the sticky flop, and logic is also present for the 'WIC' property. This is customarily a single bit field.
- SOR: set on read – the field is set to all 1's when the address is read.
- DOR: decrement on read – the field is decremented by 1 when the address is read.

- DORS: decrement on read, saturating – if not zero, the field is decremented by 1 when the address is read.
- IOR: increment on read – the field is incremented by 1 when the address is read.
- IORS: increment on read, saturating – if not all 1's, the field is incremented by 1 when the address is read.
- W1S: write-1-to-set – the field is set to 1's when the address is written and the corresponding bit(s) is/are 1, but is not altered if the bit(s) is/are 0. (csrGen implements this on a per-bit basis for multi-bit fields).
- WO: write only – the field is only present in the write logic.
- ST0: sticky low – like sticky, but the sticky value stays 0 if the nominal value is ever 0.
- Incr: incrementer – the field is an incrementing counter. The name in the definition is taken as a one bit input to the module (unless 'Intern'), and a counter with the field width is defined with '_cntr' appended to the name.
- IncrS: incrementer, saturating – same as Incr, but stops incrementing at all 1's.
- Decr: decrementer – similar to Incr, but decrementing.
- DecrS: decrementer, saturating – same as Decr, but stops decrementing at all 0's.
- SUB/SUBM: subset and subset msb – if several fields/addresses are to be catenated to form a larger field, the same name is used for each subset and is defined as a field with the SUB property followed by a range definition. The subset field with the most significant bit of the larger field must use the SUBM property. For example:


```

%A 0
31:0 bigfield SUB 31:0
%A 1
15:0 bigfield SUBM 47:32

```
- Shadow: any field that is not to be constructed by csrGen, but is implemented by user added logic can still be defined in the template for the purpose of documentation and generating firmware or verification definition files. The shadow property prevents csrGen from generating logic for the register.
- Pulse: the field asserts for one cycle when written as 1, and always reads as 0.
- PulseA: assert until acknowledged – when written as 1, the field asserts until an acknowledgement is received. The name used for the acknowledgment is the field name with `_ack` appended. The acknowledgment results in the deassertion at the next clock edge.
- any numeric value is taken as a reset value for the flops in the register.
- buss: for `%AREPEAT` only, single bit fields only, specifies that the name is taken as a vector with a width matching the repeat count, and each address corresponds to one bit of the vector.

4.5 csrGen Module Properties

A number of options can define properties of the entire module created by csrGen.

- `%B <name>` specifies a name to be used for the verilog module and files. If not specified, the default value of `chip_up_ifc` is used.
- `%C <name>` specifies a name for flop clocks. It is added to the module inputs. The default name is `clock`. The clock is used as a positive edge.

- `%RST <name>` specifies a name for flop resets. It is added to the module inputs. The default name is `init1`. The reset is used as a negative (logic 0) asserted value.
- `%WD <name>` specifies the name used as the source of data written to CSRs. The default name is `up_datain`. This is not automatically added to the module inputs, so may be an internal signal, or declared with a `%I` declaration.
- `%RD <name>` specifies the name used as the destination for data read from CSRs. The default name is `up_dataout_D`. This is not automatically added to the module outputs.
- `%RM <number>` specifies that the read multiplexer should be partitioned into blocks of addresses – the number is the size of each block. This is done by appending a number to the read data destination name – the number at each address is calculated by dividing the address by the read multiplexer block size. Using this option, the designer can create a pipelined read multiplexer (`csrGen` creates just the logic up to the first pipeline stage).
- `%AUTO` selects the use of 'automatics' from emacs verilog mode for the module IO list and for the sensitivity list. `csrGen` will execute emacs to expand these automatically.

4.6 Flop Properties

For flops created from the `%F` and `%OF` structures and as CSR fields, the reset and direct values do not have to be specified on the declaration line, they can be set separately with `%RESETVALUE` and `%FLOPVALUE`, with the flop name and value as parameters. `%RESETVALUEREPEAT` and `%FLOPVALUEREPEAT` are equivalents for repeating structures, taking a repeat count after the flop name.

When declared, a direct value for a flop is used instead of `<name>_D` as the right side of the flop posedge assignment.

4.7 Direct Verilog Code

Two definitions are used with `csrGen` for placing verilog code into the template file that will be used in the module verilog code that is generated: `%VCL` and `%V`. Both mark the start of a block of verilog code (many lines) that ends with a `%E` (which is required).

`%VCL` is always necessary for the designer to control when the write and read operations are performed, and is for combinational logic statements (in the form of verilog's blocking assignments). Code between the `%VCL` and the `%E` is preceded by an `always @ (...) begin` and default assignments for all `_D` inputs to flops, and followed by `end`.

Code in the `%V` block is enclosed only by the module declaration and the `endmodule` statement. This code follows all of the verilog declarations (input, output, wire, reg). This may be used for instantiations, assign statements, and even additional always blocks as desired by the designer.

Both `%VCL` and `%V` allow for a line (or several lines) to be repeated with any `%` characters in the line(s) being replaced by an incrementing index (the elusive verilog generate statement). This is

done with the `%LOOP` and `%LOOPEND` declarations. If `%LOOP` has one parameter, it is a repeat count; if there are two parameters, they are the first and last indexes.

Additionally, any lines with verilog format comments (`//` or blocks formed by `/*` and `*/`) are copied to the verilog module file (before the module).

5 Design Verification

csrGen aids design verification by automatically creating definitions associated with every CSR field, specifying the address, width, and bit position.

Another structure often required for verification is coverage measurements. csrGen can be enhanced to generate code to measure read and write coverage of CSR fields. Today, this may be in formats/structures that differ between projects, but perhaps a general structure for this will emerge so that it can become a standard feature of csrGen.

The template file does specify CSR behavior well enough to lead to automatic testing of many CSR operations. This could include:

- basic read and write operations
- randomized data to validate field position and independence
- reset value checking
- clear-on-read, write-1-to-clear testing

6 Future Enhancements and Improvements

6.1 csrGen Enhancements

csrGen has many places for improvement. This section is a loosely arranged laundry list.

The template file parsing is simplistic (a hardware engineer rather than a ‘proper programmer’ developed it). Much of the verilog creation is currently done as the template file is parsed, so no main data structure results from the parsing. That sort of data structure may be needed to enable future enhancements.

Wider exposure of csrGen may bring out requests or completed code for additional features and logic structures, as each design may have different requirements.

A precursor to csrGen produced documentation in the form of tables describing the contents of each address (in Microsoft's RTF format) – this is very desirable (HTML seems like the right format now).

The combinational logic block coding structure may be criticized as possibly inefficient in simulation, as the block may be invoked multiple times in one clock cycle. It may be possible to use that structure only for fields that the designer wants to manipulate with added logic, and place the plain fields in logic that is directly in the clocked always block.

Some designs may incorporate multiple processor interfaces, requiring multiple address maps (with some shared CSRs – various sharing mechanisms are possible).

Support for multiple clocks (specifically for %F flops) may be desirable for some designs, although I expect that all writeable CSRs in a single address map would be on a single clock.

Another structure that could be added would be 'flopped constants', since some constants that may be used as CSRs are values like version numbers, and if a chip is ECO'ed, the version number should usually also be changed. It can be easier to change if the constant value is at the input to flops rather than absorbed directly into read multiplexers – and using the complement of the value as the reset value can prevent synthesis from optimizing away the flops.

6.2 Other Directions

csrGen has been developed as a single purpose template expansion tool, but the ideas embodied in it may suggest constructs for other languages or tools. Could the emacs verilog mode be enhanced to expand a set of CSR definitions found inline? Could a higher level hardware design language allow users to specify CSRs in a manner like csrGen by building the structures from an object class?

7 Conclusions and Recommendations

I believe that csrGen has saved me enough time and mistakes to balance the investment I've made in developing it. csrGen has been part of the design flow for two ASICs I have developed. Each design had specific needs that led to new features.

csrGen is free for any user, and I hope that it may be adopted for use anywhere designers are implementing large memory maps of CSRs. I welcome feedback, whether as code contributions or requests for more features. As of the end of December 2002, the csrGen PERL script has been downloaded from my website over 800 times since it was mentioned in an EE Times article in September 2002, but to date I've only received feedback from one user, who suggested further CSR structures (which I added) and contributed code to add support for the emacs verilog mode.

RTL code for CSRs can be fruitfully automated, and csrGen is a good illustration of how it can be done.

8 Acknowledgements

Thanks to Mark Matulaitis, Mark Levesque, Andy Moroney, Jeff Koehler, Jim Wu, George Nicholas, Chris Payson, and Steve Sherman for ideas and suggestions that have ended up in various ways in csrGen.

9 References

[1] Verilog mode for Emacs, verilog-mode.el, 1996-2002 (and continuing), Mike McNamara, <http://www.verilog.com/verilog-mode.html>