505 Registers or Bust

Wilson Snyder

Nauticus Networks

wsnyder@world.std.com

ABSTRACT

This paper discusses how to implement and verify I/O CSRs. First, Verilog code for implementing CSRs is examined. Several improvements of the RTL are made, to improve readability, maintainability, and gate area.

Next, we discuss direct reads and writes, a simple technique for vastly speeding up initialization of the CSRs.

The remainder of the paper discusses Vregs, the author's public domain program. Vregs converts HTML documents from Microsoft Word or Adobe Framemaker to Verilog defines, C defines, C++ classes and automatic tests.

1 Introduction

Most chips have software programmable I/O CSRs, and everyone seems to have a different way of implementing them. On a recent chip of mine with 505 unique CSRs, we found religion. By standardizing our RTL coding and documentation format, we automatically extract all CSR information from the document, greatly simplifying CSR usage for both the hardware designers and software users of the device.

2 What is a CSR?

In this paper, I'll be talking about Control and State Registers (CSRs). These are a memory mapped location that software can read and write to communicate with the device. They are also sometimes called I/O registers, Configuration registers, or programmable state registers. Each CSR can have 32 bits (or some other constant), each bit of those bits may be part of a field that affects the hardware.

CSRs are one of the key ways software and hardware interact: there are many chips can be fully programmed only by understanding the CSRs of the device. Having an elegant, consistent and fully tested CSR set is often one of the first deliverables from a hardware team to a software team. An improvement in CSR design thus reduces development time, and turn time.

3 Verilog Implementation

The primary audience for this paper is hardware designers. Thus, it's wise to talk about Verilog code for a set of CSRs, and how it can be improved. Software designers can ignore this section, and VHDL addicts can translate.

Let's look at very simple code that implements 2 CSRs. We'll make the first CSR 30 read-write bits (bit 31 is zero). We'll make the second CSR 2 bits, each bit of which has a separate function.

This code is fairly simple; it has an address bus with write data and strobe, and is always reading. (This code uses Verilog-Mode for Emacs, as it shrinks the size of the code. It available from the author's website.)

```
module regex_1 (/*AUTOARG*/);

   input     clk;
   input     reset_;    // Synchronous reset

   // I/Os from the CPU or whatever bus needs to read/write the regs
   input  [31:2] io_address;  // I/O Address bus (byte address)
   input  [31:0] io_wdata;    // I/O Write data bus
   input         io_wstrb;    // I/O Write strobe pulse
   output [31:0] io_rdata;    // I/O Read data bus

   // The I/O CSR values
   output [30:0] funcone_r;   // I/O bit field 1 (at address 0)
   output    functwo_r; // I/O bit field 2 (at address 4[1])
   output    functhr_r; // I/O bit field 3 (at address 4[0])
```

```
   /*AUTOREG*/

   //==== Writing I/O regs

   always @ (posedge clk) begin
      if (~reset_) begin
       funcone_r <= 31'h0;
       functwo_r <= 1'b0;
       functhr_r <= 1'b0;
      end
      else begin
       if (io_wstrb && {io_address[31:2],2'b00} ==
             32'h0000_0000) begin
          funcone_r <= io_wdata[30:0];
       end
       if (io_wstrb && {io_address[31:2],2'b00} ==
             32'h0000_0004) begin
          functwo_r <= io_wdata[0];
          functhr_r <= io_wdata[1];
       end
      end
   end

   //==== Reading I/O regs

   always @ (/*AS*/) begin
      casex ({io_address[31:2],2'b00})
      32'h0000_0000: io_rdata = {1'b0, funcone_r};
      32'h0000_0004: io_rdata = {30'b0, functhr_r, functwo_r};
      default:       io_rdata = 32'h0; // Illegal address
      endcase
   end

endmodule
```

3.1 Removing Constants

Regex_1 is a fine start, and the sort of code you can find in many chip designs. However, there is quite a bit of room for improvement, even with this simple code.

First, consider what happens when we change the address or bits that a field uses. The code has magic numbers all over the place, and we can't even grep (search) to find where a specific CSR is used. (This follows the general coding rule of using defines instead of “Magic Numbers;” see any good programming style book, or http://www.cs.umd.edu/users/cml/cstyle/indhill-cstyle.html.)

To fix this, let's have a set of defines for all the constants. We'll prefix our defines with RA_ for Register Address, RB_ for the bit number the CSR bit begins at, and RE_ for the bit number the CSR ends at. I haven't used the defines in the code that reads the CSRs yet, we'll get to that in a moment.

```
module regex_2 (/*AUTOARG*/);

   `include "same_decls_as_regex_1.v"

   //==== Writing I/O regs
```

```
`define RA_REGA         32'h0000_0000     // Address of CSR A
`define RB_REGA_FUNCONE 0     // Bit # funcone field begins
`define RE_REGA_FUNCONE 0     // Bit # funcone field ends
`define RA_REGB         32'h0000_0004     // Address of CSR B
`define RB_REGB_FUNCTWO 0     // Bit # functwo field begins
`define RE_REGB_FUNCTWO 0     // Bit # functwo field begins
`define RB_REGB_FUNCTHR 1     // Bit # functhree field begins
`define RE_REGB_FUNCTHR 1     // Bit # functhree field begins

   always @ (posedge clk) begin
      if (~reset_) begin
       funcone_r <= 31'h0;
       functwo_r <= 1'b0;
       functhr_r <= 1'b0;
      end
      else begin
       if (io_wstrb && {io_address[31:2],2'b00} == `RA_REGA) begin
          funcone_r   <= io_wdata[`RE_REGA_FUNCONE:`RB_REGA_FUNCONE];
       end
       if (io_wstrb && {io_address[31:2],2'b00} == `RA_REGB) begin
          functwo_r   <= io_wdata[`RB_REGA_FUNCONE];
          functhr_r   <= io_wdata[`RB_REGA_FUNCONE];
       end
      end
   end

   //==== Reading I/O regs

   always @ (/*AS*/) begin
      casex ({io_address[31:2],2'b00})
      `RA_REGA: io_rdata = {1'b0, funcone_r};
      `RA_REGB: io_rdata = {30'b0, functhr_r, functwo_r};
      default:  io_rdata = 32'h0;
      endcase
   end

endmodule
```

3.2 Loading all bits

Having to manually look at which bits are used makes it hard to determine the layout of the CSRs. In the next example, I've formed a define with the name of the wires containing the state to be read or written. With this technique, you can add or change the size of a field and not even have to touch the read or write code.

If a bit is unused, I simply refer to a vector called "unused". This is set to zero at the beginning and end of the statement so that Synopsys Design Compiler will completely optimize away the "unused" signal.

```
module regex_3 (/*AUTOARG*/);

   `include "same_decls_as_regex_2.v"

   //==== Writing I/O regs

`define RWIRES_REGA     {unused[31], funcone_r[30:0]}
`define RWIRES_REGB     {unused[31:2], functhr_r, functwo_r}
```

```
   reg [31:0] unused;
   always @ (posedge clk) begin
      if (~reset_) begin
       funcone_r <= 31'h0;
       functwo_r <= 1'b0;
       functhr_r <= 1'b0;
      end
      else begin
       if (io_wstrb && {io_address[31:2],2'b00} == `RA_REGA) begin
          `RWIRES_REGA <= io_wdata;
       end
       if (io_wstrb && {io_address[31:2],2'b00} == `RA_REGB) begin
          `RWIRES_REGB <= io_wdata;
       end
      end
      unused <= 32'h0;
      // Unused prevents needing any flops to implement unused bits
   end

   //==== Reading I/O regs

   always @ (/*AS*/) begin
      unused = 32'h0;   // Insure that unused bits return zeros
      casex ({io_address[31:2],2'b00})
      `RA_REGA: io_rdata = `RWIRES_REGA;
      `RA_REGB: io_rdata = `RWIRES_REGB;
      default:  io_rdata = 32'h0; // Illegal address
      endcase
   end

endmodule
```

3.3 Area for Reset

There are two final improvements for the physical design side.

When synthesized, we're adding a lot of logic to reset the CSRs, as every bit of every CSR on the chip needs a AND gate (or similar) to zero the CSR. We can do better; simply zero the write data during reset, and force all CSRs "open". This is vastly more efficient in space, and can save a level of logic too. An added benefit is we no longer need the if statement that resets the CSRs; this eliminates the potential bug of forgetting to reset a CSR because it wasn't added to the reset part of the if statement.

In the next example below, I've added the zeroing of the input data during reset, but normally it would be done at the "top" of the chip. I've also added the loading of CSRs during reset, and use a define to make it more clear.

One caution; this technique presumes you’re already using a synchronous reset strategy. Synchronous resets can be much more area efficient, but can result in X propagation during gate simulations. We got around the X propagation problem by doing three types of simulations, one with all X’s initialized to 0, one with all X’s initialized to 1, and one with X’s randomized. This conversion of X’s to a two state simulation is further described by the author in

http://www.deepchip.com/items/0342-08.html. (This works; the author's chips have never had a reset bug.)

3.4 Area for Interconnect

Across the whole chip, we'll probably have many CSR modules like this one. All the modules will then be interconnected and something will mux between their read data. This makes a routing nightmare.

Instead, let's have each module return the write data when a read of an unknown address comes in. Then we can simply connect one module's write data to the previous module's read data. This chain results in half the number of wires, and converts all busses to point-to-point interconnects.

You do have to be a little careful with daisy-chaining, as it can potentially lead to some nasty post-layout timing problems. To avoid these, the daisy-chained CSR blocks should be interconnected in the physical order that they reside around the die, to reduce routing and cross chip paths. To further improve the paths, multiple daisy chains can be used on larger chips, just MUX the final stage of each chain. Finally, devices that only do infrequent CSR accesses can make the data-bus a multicycle path, with the appropriate change in the control logic.

The only change to implement daisy-chaining in the example is in the default of the read case statement.

```
module regex_4 (/*AUTOARG*/);

   `include "same_decls_as_regex_2.v"

   //==== Writing I/O regs

`define RWIRES_REGA     {unused[31], funcone_r[30:0]}
`define RWIRES_REGB     {unused[31:2], functhr_r, functwo_r}

   // Form write data with zeros during reset.
   // In real applications, the logic generating io_wdata
   // should do it
   wire [31:0] io_wdata_zrst = (~reset_) ? 32'h0 : io_wdata;

   // Macro for checking if we need to write the CSR at
   // the given address.  We write during reset to load the
   // zeros in io_wdataz;
`define WRREGTEST ~reset_ || io_wstrb && {io_address[31:2],2'b00} ==

   reg [31:0] unused;
   always @ (posedge clk) begin
      if (`WRREGTEST `RA_REGA) begin `RWIRES_REGA <= io_wdata_zrst; end
      if (`WRREGTEST `RA_REGB) begin `RWIRES_REGB <= io_wdata_zrst; end
      unused <= 32'h0;  // Avoid flops to implement unused bits
   end

   //==== Reading I/O regs

   always @ (/*AS*/) begin
      unused = 32'h0;   // Insure that unused bits return zeros
```

```
      casex ({io_address[31:2],2'b00})
      `RA_REGA: io_rdata = `RWIRES_REGA;
      `RA_REGB: io_rdata = `RWIRES_REGB;
      default:  io_rdata = io_wdata;  // Chain from other module
      endcase
   end

endmodule
```

3.5 Direct Access

You've got all of your CSRs implemented in RTL code, you've tested them all, and now it comes time for real testing. You're probably going to find it takes a lot longer to initialize the chip then to run the test.

Our chip with 505 CSRs had not 505 memory locations to be initialized but about 48K locations, or 190KB. This is because some of the CSRs are not single locations, but arrays (memories). For example, several of these 505 CSRs had 8192 entries. Writing this 190KB of data takes quite a bit of time. On a fully optimized VCS simulation, this initialization took us a minimum of 20 minutes. This is obviously unproductive time, as the CSRs have already been tested for read/write and addressing, and you don't need to test initialization every simulation run.

What we would like is a way to read and write CSRs in zero time, and use that in tests. This reduces that 20 minute simulation initialization to about 45 seconds.

Below we've taken our example and added two routines for direct reading a longword and direct writing a longword. Somewhere in our test bench we then add a master routine that understands the entire address map, and calls the direct_read or direct_write function in the appropriate submodule. As this is the last example, the whole code is reproduced for comparison with the original.

```
module regex_final (/*AUTOARG*/);

   input     clk;
   input     reset_;    // Synchronous reset

   // I/Os from the CPU or whatever bus needs to read/write the regs
   input  [31:2] io_address;  // I/O Address bus (byte address)
   input  [31:0] io_wdata;    // I/O Write data bus
   input         io_wstrb;    // I/O Write strobe pulse (latch addr/data)
   output [31:0] io_rdata;    // I/O Read data bus

   // The I/O CSR values
   output [30:0] funcone_r;   // I/O bit field 1 (at address 0[30:0])
   output    functwo_r; // I/O bit field 2 (at address 4[1])
   output    functhr_r; // I/O bit field 3 (at address 4[0])

   /*AUTOREG*/

`define RA_REGA         32'h0000_0000     // Address of CSR A
`define RB_REGA_FUNCONE 0     // Bit # funcone field begins
`define RE_REGA_FUNCONE 0     // Bit # funcone field ends
`define RA_REGB         32'h0000_0004     // Address of CSR B
`define RB_REGB_FUNCTWO 0     // Bit # functwo field begins
```

```
`define RE_REGB_FUNCTWO 0     // Bit # functwo field begins
`define RB_REGB_FUNCTHR 1     // Bit # functhree field begins
`define RE_REGB_FUNCTHR 1     // Bit # functhree field begins

   //==== Writing I/O regs

`define RWIRES_REGA     {unused[31], funcone_r[30:0]}
`define RWIRES_REGB     {unused[31:2], functhr_r, functwo_r}

   // Form write data with zeros during reset.
   // In real applications, the logic generating
   // io_wdata should do it
   wire [31:0] io_wdata_zrst = (~reset_) ? 32'h0 : io_wdata;

   // Macro for checking if we need to write the CSR
   // at the given address.  We write during reset to load
   // the zeros in io_wdataz;
`define WRREGTEST ~reset_ || io_wstrb && {io_address[31:2],2'b00} ==

   reg [31:0] unused;
   always @ (posedge clk) begin
      if (`WRREGTEST `RA_REGA) begin `RWIRES_REGA <= io_wdata_zrst; end
      if (`WRREGTEST `RA_REGB) begin `RWIRES_REGB <= io_wdata_zrst; end
      unused <= 32'h0;  // Avoid flops to implement unused bits
   end

   //==== Reading I/O regs

   always @ (/*AS*/) begin
      unused = 32'h0;   // Insure that unused bits return zeros
      casex ({io_address[31:2],2'b00})
      `RA_REGA: io_rdata = `RWIRES_REGA;
      `RA_REGB: io_rdata = `RWIRES_REGB;
      default:  io_rdata = io_wdata;  // Chain from other module
      endcase
   end

   //==== Direct access
`ifdef synopsys `else   // Ifdef is better then translate on/off as it
                        // allows simulation of the synthesis code, and
                        // the if/endif insures matching on/off pairings.
   task direct_write;
      input [31:0] address;
      input [31:0] wdata;
      reg [31:0] unused;
      begin
       unused = 32'h0;
       case (address)
         `RA_REGA: `RWIRES_REGA <= wdata;
         `RA_REGB: `RWIRES_REGB <= wdata;
        default: $display("%%Error: Bad direct_write address %x\n",
                          address);
       endcase
      end
   endtask

   task direct_read;
      input [31:0] address;
      output [31:0] rdata;
      begin
       case (address)
```

```
            `RA_REGA: rdata = `RWIRES_REGA;
            `RA_REGB: rdata = `RWIRES_REGB;
            default: $display("%%Error: Bad direct_read address %x\n",
                      address);
          endcase
        end
      endtask
    `endif

    endmodule
```

Note there is some replicated code between direct_read and normal read. This can be removed by using a function, though you need to be careful that the sensitivity list includes the data bits inside the function. An easy way to solve this is to have the reading be in a posedge clk block instead of as combinatorial logic.

This same zero time direct read and write can be extended to allow software groups to use it, also speeding up their tests. This technique can also be used in gate simulations, though you need to figure out the path into each storage flop for each synthesized CSR bit.

4 From the spec!

We've simplified the RTL code, but there's still the step of producing all those defines for the CSR bits and addresses. We could continue to do this manually, but what happens when we hand it off to the software team? They're going to need all of their own defines for their C code. We'd thus like to generate all of the constants with a program.

The natural question is where to get the information on the layout of the CSRs for the program. There is one place where the information already exists, the specification.

So, we'll read the specification and create the defines automatically.

Some teams have also automated this process in the opposite direction; have a ASCII file describing the CSRs which then creates the documentation. Personally, I'm a fan of creating documentation automatically and text formatting languages, but unfortunately most people like WYSIWYG editors, especially technical writers. Thus I use the documentation as the reference.

4.1 Vregs

Vregs is a program the author wrote that does this documentation conversion. First, you write your specification using the examples in the appendix (Vregs only cares about the general table layout, not style and font issues.) Almost any program will do, both Microsoft Word and Adobe Framemaker have been used in the past.

When the document is ready for conversion, save the document as HTML. Microsoft Word annoyingly opens the HTML after saving; be sure to exit the HTML version immediately.

You then run Vregs, which reads the HTML code and writes out a .vregs file, which contains a summary of the extracted information. Vregs also writes out headers with #defines for all of the constants and magic numbers in the spec. Vregs also has a mode to read the .vregs file directly, which is useful for archiving the register definitions independently of the specification document itself.

5 Vregs Outputs

This chapter shows some of the outputs from the Vregs program. This output was derived from this exact document; in this case all of the relevant information is in the appendixes, though Vregs doesn't care where it is in the document.

5.1 .vregs file

The first output from Vregs is a .vregs file. This file summarizes the results of the HTML file, and is useful for archiving the results of the specification. Below is part of the vregs_spec.vregs file that results from running this document through Vregs. Note how the information in the .vregs file has reduced the pages of specification to only the most basic information needed for the CSRs.

```
package vregs_spec
  reg   R_ExReg1  VregsExReg1 0x18FFFF0000
   type R_ExReg1
     bit     LastCmd   31:28  RW  ExEnum    X  "Enumerated field"
     bit     ReadOnly  20     R   bool      X  "Read Only Bits"
     bit     LowBits   3:0    RW  uint32_t  0  "Random Low Bits"
  // Enumerations
  enum  ExEnum
     const   ONE               4'b0001   "Command One"
     const   TWO               4'b0010   "Command Two"
     const   FIVE              4'd5      "Command Five"
     const   FOURTEEN          4'he      "Command Fourteen"
  // Defines
     define  CMP_DEFINED_FOOD  48'hfeed  "Definition of Food"
     define  CMP_DEFINED_ONE   4'd1      "Definition One"
```

5.2 Verilog Header

Vregs makes several header files for different languages. For Verilog, the header file includes all the define values, enumeration values, plus defines for every CSR address and CSR bit position.

Below is part of the vregs_spec_defs.v that results from this document. The CMP_ definitions come almost exactly from the definition section in appendix B. The address of ExReg1 becomes a RA_ definition, and the bit numbers of the fields of ExReg1 become CR, CB and CE defines. (Standing for class range, class begin, and class end, respectively.)

```
`define CMP_DEFINED_FOOD         48'hfeed // Definition of Food
`define CMP_DEFINED_ONE              4'h1 // Definition One

`define RA_ExReg1          40'h18FFFF0000 // Address of R_ExReg1
`define CR_ExReg1_LastCmd           31:28 // Field Bit Range: 31:28
`define CB_ExReg1_LastCmd              28 // Field Start Bit: 31:28
`define CE_ExReg1_LastCmd              31 // Field End Bit:   31:28
```

5.3 C Header

Just as Vregs produces a Verilog header, it also produces a C header. All the same values are there, just the syntax of the file is different. Likewise Vregs produces a Perl header, and can easily be modified to produce a header for VHDL or any other language.

```
#define CMP_DEFINED_FOOD         0xfeedULL /* Definition of Food */
#define CMP_DEFINED_ONE                1  /* Definition One */

#define RA_ExReg1          0x18FFFF0000ULL /* Address of R_ExReg1 */
#define CR_ExReg1_LastCmd           31:28  /* Field Bit Range: 31:28 */
#define CB_ExReg1_LastCmd              28  /* Field Start Bit: 31:28 */
#define CE_ExReg1_LastCmd              31  /* Field End Bit:   31:28 */
```

5.4 C Information & Testing

It's quite common to have logfiles and similar debugging information that print the address of a transaction in the design. Many of these messages print the address of a CSR, and data being read or written. Rather then having to decode what the address means in your head, Vregs has all of the information to annotate the logfiles with the symbolic name of the CSR.

Vregs produces a vregs_spec_info.cpp file, which with a little stub routine allows C++ to get the ASCII symbolic name of any address. With a little PLI wrapper similar to the following example, this same function can be provided to Verilog.

```
// In Verilog
reg [31:0] adr = 32'h12340000;
$write (" Address = %x, Register Name = ");
$write_reg_name(adr);   $write ("\n");

// In C++ code for PLI library
#include "VregsRegInfo.h"
VregsRegInfo reginfo;
void call_this_at_program_startup (void) {
      vregs_spec_info::add_registers(&reginfo);
}

void write_reg_name_pli (void) {
      unsigned int   addr = tf_getp(1);
      const char* name = reginfo.addr_name(addr);
      if (name) io_printf ("%s", name);
}
```

This information also contains information on which bits in each CSR are readable, writable, and read/write. This allows a for() loop in a test to use this information and automatically test for proper implementation of every CSR.

5.5 C Class File

Vregs also makes C++ classes for every CSR. This makes it very easy to access the bit fields in a CSR. In this example R_ExReg1 is the class created by Vregs. SIZE is a constant representing the size of the CSR, and lastCmd is a bit field that Vregs has defined as an accessor method of R_ExReg1.

```
R_ExReg1  regdat;
```

```
read_memory (RA_ExReg1, &regdat, regdat.SIZE);
regdat.lastCmd (new_value_for_lastCmd_field);
write_memory (RA_ExReg1, &regdat, regdat.SIZE);
```

Some complex projects have more complex interactions with software then just CSRs. Vregs allows for the specification to contain Classes that are not associated with CSRs. The bits of these classes are accessible as if they are CSRs to hardware, and as C++ classes to software. For example, if you have a TCP/IP frame format defined, software can treat it as a large multiword structure, and hardware can get the defines for the bit layouts of the structure.

Vregs also understands the types of each field, so that you can reference a enumeration declared in the document, and C++ will insure that access to that field is only done through the enumeration.

6 Conclusion

This paper has presented several ideas for improving CSR RTL coding, for speeding up CSR simulation times, and for automatic extraction of registers from specifications. With these techniques and scripts, you can automate most of the work in adding and verifying new CSRs, and lead to documentation that is proven to accurately represent the design.

7 Obtaining Vregs

Vregs is a public domain tool that may be downloaded off the author's site at http://veripool.com. As a perl package, it is also available from all CPAN (Comprehensive Perl Archive) sites.

To contact the author directly, or report bugs, email Wilson Snyder wsnyder@wsnyder.org or wsnyder@world.std.com.

A Example Definitions Section

These appendixes describe the register layouts that the Vregs program understands. For complete documentation, see the distribution.

This first appendix shows the layout of constant definitions that the Vregs program parses from this document.

A.1 Description

A definition section starts with the word "Defines" alone on a line. Vregs will create #define statements for each mnemonic/constant pair.

There are three columns for each value in the enumeration. Columns can be in any order.

The constant column is the value for the define. Numeric values are in Verilog format, with the width and a h/d/b for hex, decimal and binary, respectively. String and other formats are not supported yet.

The mnemonic must be all upper case.

If constants are being defined for a series of values, a enumeration is probably a better way to do it.

A.2 Example Definitions

Description

This table shows an example definition table. The information in the header below is prepended with a underscore to all mnemonics, to prevent the global name space of defines from causing trouble

Defines

CMP

Constant	Mnemonic	Definition (header comments in parenthesis)
4'd1	DEFINED_ONE	*Definition One*. Text up to the first period will be annotated into the output files.
48'hfeed	DEFINED_FOOD	*Definition of Food*.

B Enumerations

This appendix shows an example enumeration definition that the Vregs program parses from this document.

B.1 Description

A enumeration triggers off from the word “Enum” alone on a line. Vregs will create a C++ enumeration for the values, and Verilog #defines for each of the values in the enumeration.

There are three columns for each value in the enumeration. Columns can be in any order.

The constant column is the value for the mnemonic. Values are in Verilog format, with the width and a h/d/b for hex, decimal and binary, respectively.

The mnemonic must be all upper case. Underscores are acceptable, but strongly discouraged.

A table row may have a empty mnemonic column if the definition contains the text “reserved”.

B.2 Example Enumeration (This header is ignored)

Description

This table shows an example enumeration table.

Enum

ExEnum

Constant (comments)	Mnemonic	Definition (header comments in parenthesis)
4’b0000		*Reserved*
4’b0001	ONE	*Command One*. Text up to the first period will be annotated into the output files.
4’b0010	TWO	*Command Two*.
4’d5	FIVE	*Command Five*. Number in decimal.
4’he	FOURTEEN	*Command Fourteen*. Number in hex.

C Class Definitions

This appendix shows an example Class definition that the Vregs program parses from this document.

C.1 Description

Class Declarations key off the word “Class” alone on a line. Class names must begin with C_. The C_ prefix will be stripped for the real name of the class.

A header called “Attributes” before the header specifies special actions for the class.

Attribute	Description
-variablelen	The class is of variable length, with data words appended to the end of the structure.
-netorder	The structure longwords are in network order. (Big endian).
-noarray	For CSR types, declares the region as consisting of raw bytes, rather then a array of memory. Thus when asking what the name of a address within the space is, the return will be something like “Ram+0x10” rather then “Ram[4]”.

There are five columns for each value in the enumeration. Columns can be in any order.

The bit column defines which bits the field occupies. Bit numbers are then expressed in MSB:LSB order. Bits can be of any width; there is no restriction of their being less then 32. For readability w#[] indicates the bits in the brackets are in a given 32-bit word, 32 times the # will be added to the bit numbers in the brackets. For example w3[10] is bit 10 in longword 3, equivelent to writing [106] (3*32+10 = 106). Fields may consist of multiple disjoint segments, separated by commas. (w0[15:13] or w0[15,14,13] or w0[15],w0[14],w0[13] are all equivelent.)

The mnemonic must begin with a capital, and contain no underscores.

The type is the C++ type of the field. If left blank, single bit fields will be assumed to be “bool” and multiple bit fields will be unsigned integers. The entire column may be deleted if the default is acceptable for all of the fields.

The constant column is used to specify the given bit range always contains a certain value. Often it will be expressed as a enumeration, such as when specifing a command number field inside the layout of one specific command.

C.2 Example Base Class

Description

This is an example base class definition. In this example, we'll define ExBase which is a generic format of a message, then we'll define specific messages.

Class

ExBase

Attributes

-netorder

Bit	Mnemonic	Type	Constant	Definition
w0[31:28]	Cmd	ExEnum		*Command Number*. Encoding is in the ExMnem table. You'll see in later derived classes how this specifies which command this generic class represents.
w0[28]	CmdAck			*Command Needs Acknowledge. Overlaps Cmd.* This is a Boolean field, since it's one bit and has no specified type. The "overlaps Cmd" field turns off the normal warning that bit 28 is used twice. This is done in this example as one bit of the command always indicates a specific piece of information that we want to extract.
w0[27:24]	FiveBits			*Five Bits.* This field will become a unsigned int.
w0[15:0], w1[31:0]	Address	Address		Address. This field spans two words to make a wide 48-bit field.

C.3 Example Class

Description

This shows a class which inherits the base class defined earlier.

Class

ExClassOne : ExBase

Bit	Mnemonic	Type	Constant	Definition
w0[31:28]	Cmd	ExEnum	ONE	*Command Number*. This field is a constant, as it indicates that this is message type ONE.

C.4 Another Example Class

Description

This is another example class.

Class

ExClassTwo : ExBase

Bit	Mnemonic	Type	Constant	Definition
w0[31:28]	Cmd	ExEnum	TWO	*Command Number*. Indicates the second command.
w0[27:24]	FiveBits			*Five Bits.* You can redeclare fields in the base class, but they must have the same name and mnemonic.
w2[31:0]	Payload			Another field that this message tacks onto the end of the base class.

D CSR Declarations

This appendix shows an example register definition that the Vregs program parses from this document.

D.1 Definition

CSR declarations key off the word “Mnemonic” alone on a line. CSRs always start with R_, then a name beginning with a capital, and containing no additional underscores. They can also have attributes like the Class Declarations.

The address is specified in hex, with a leading 0x. The maximum width of the address is coded into the vregs program; it defaults to 48 bits, which is sufficient for most projects.

The table indicates the bit layout:

The Bit column is the bits the field occupies.

The Mnemonic is the name of the bit field. It begins with a underscore, then a capital. There must be no other underscores in it.

Reset indicates the value after chip reset. X or N/A indicates not reset. FW0 or FW-(some perl expression) indicates that the value is loaded in by firmware during initialization.

Type indicates the C++ type of the field. If unspecified, bool is used for 1 bit entries, else uint32_t is used.

Access indicates read/write, read-only, etc. Read side effects indicate that reading the CSR can change the value in the CSR or cause other effects. Write side effects indicate that changing the value may change other CSRs, and is only used to tell the CSR testing to skip writing this CSR.

Access	**Read Action**	**Write Action**
RO	Read Only	Ignored
RW	Read	Write
RWS	Read	Write Side Effect
RS	Read Side Effect	Ignored
RSW	Read Side Effect	Write
RW1C	Read	Write 1 to clear
WO	Indeterminate return	Write
WS	Indeterminate return	Write Side Effect
L	Flag on any other access indicates this field needs to be written after all other CSRs. For example a unit “enable” needs to be written after all of the unit’s other CSRs are written.	

D.2 Example CSR

Description

This is an example CSR declaration.

Register

R_ExReg1

Address

0x18_FFFF_0000

Bit	Mnemonic	Access	Reset	Type	Definition
31:28	LastCmd	RW	X	ExEnum	*Enumerated field*. This field has a value represented by a complex type, in this case an enumeration**.**
20	ReadOnly	RO	X		*Read Only Bits.* This field is not writable and is not initialized during reset.
3:0	LowBits	RW	0		*Random Low Bits*. This field takes the whole CSR. As with Enums and everywhere else, only the first sentence is used to comment the output code.

D.3 Another CSR, Ranged

Description

This is another CSR, but it consists of 8 identical arrayed CSRs. The special optional comment (Add 0x10 per entry) indicates that each entry is 16 bytes apart, rather then the default dense packing of 4 bytes.

Register

R_ExRegTwo[7:0]

Address

0x18_FFFF_1000 – 0x18_FFFF_1100 (Add 0x10 per entry)

Bit	Mnemonic	Access	Reset	Type	Definition
31:0	WideField	RW	0		*Wide Field*. This field takes the whole CSR.