A Client-Server Method for Register Design and Documentation

Scott D Orangio
Manager/Principal Engineer, ADVA Optical Networking SE, 5755 Peachtree Industrial Blvd., Norcross, GA, USA, 678-728-8830 (sorangio@advaoptical.com)

Julien Gagnon
B. Ing.,Hardware Designer, Hardent Consulting, 450 rue St Pierre Suite 300, Montreal, QC H2Y 2M9, Canada, 514-284-5252 (jgagnon@hardent.com)

Abstract- For design and verification engineers, producing an error free, re-usable and testable design can be a cumbersome and time-consuming task. IP reuse and integration in chip design is growing in complexity. This paper describes a method that uses a web based client/server model to create, modify, re-use and integrate the IP register designs and documentation into a complete chip design.

I. INTRODUCTION

Our method, named 'Hdocx', uses a web interface, SQL database and python scripting to enable developers in a chip design. Using hierarchical combinations of registers and documents, which are easily combined, a chip level document, RTL and verification code outputs are created. Hdocx automates the creation of the chip register map, register testing, chip decode logic and firmware/chip integration. By automating the register and documentation functions, human errors can be detected early in the design process and developing documentation is simplified and consistent from project to project.

II. OVERVIEW

Hdocx has been developed as a working tool. The initial concept has been modified, improved and/or simplified as RTL developers applied to their designs. The parts that make the Hdocx system are shown in Figure **1**. The sections are Register Definitions (RDEF), Register Map Processing, Docx Documents, Document Processor, Database (SQL), Computer Cluster Support (Linux Server farm), Web Interface (client), Web Server, Output Files.

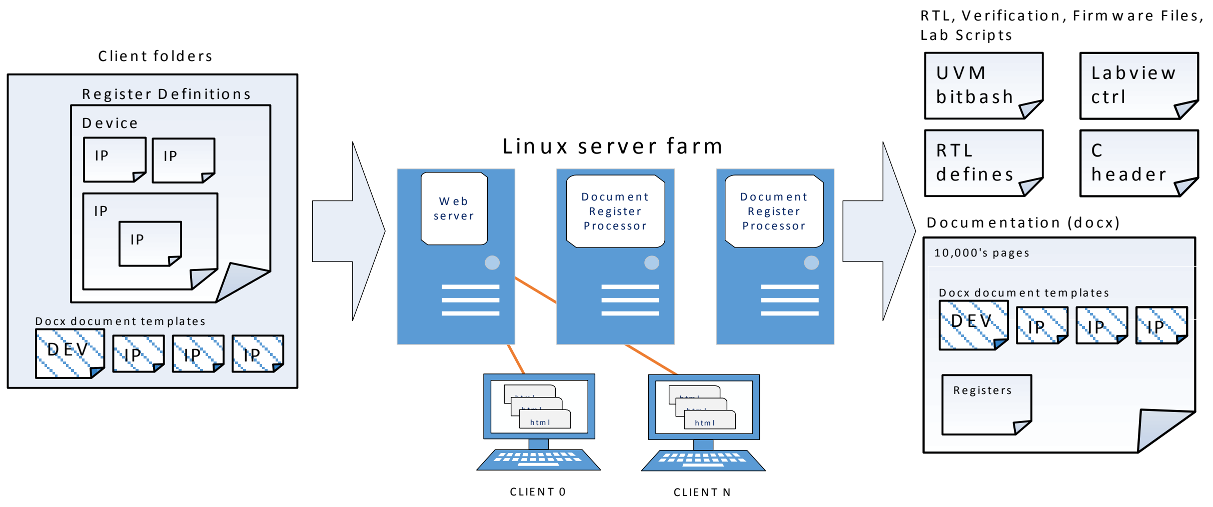


Figure 1 Hdocx System

III. Registers Definitions (RDEF)

The register definitions (in particular - registers, interrupt registers, memory blocks, and their associated bit fields and descriptions) are contained in a RDEF file. A group of RDEFs describe the functionality in the IP, device or system of devices. There is a top-level RDEF file defining the arrangement of sub-level RDEFs. All RDEFs combine to create the memory map for the system, device or IP block. It is a good practice to combine RDEF's hierarchically to parallel the hierarchical nature of chip design. It is also a good practice that RDEF files be segmented to allow for re-use. Groups of RDEFs replicate for multi-channel systems. Figure 2 is an example of rdef register description that we will use through this paper to represent this methods flow.

```
beg_block DVCON
title   ! This is the DVCON Example ip block !
notes   ! this shows rdef base syntax        !

param_name "param_dvcon_example.v"

block_byte_width        2
block_addr_width        8
block_size           0x20
block_mask            0xFF
endian               LIT

beg_reg ID         OFFSET 0x0000     DEFAULT 0xBABE
title    ! id_reg title !
id_hi         RO [15:12]         !  4bits  !
id_lo         RW [11:0]          !  12bits !
notes    ! id_reg notes !
notes    ! add description of register in this area   !
notes    ! notes are added after field definitions    !
end_reg

beg_ireg IRG_REG  OFFSET 0x04  DEFAULT 0x0000  MASK 0xFF  LEAF
title                   ! DVCON Example Interrupt !
unused            [15:5]  ! Undefined !
trans_complete_4  [4]     ! 1 = transaction_4 is complete !
trans_complete_3  [3]     ! 1 = transaction_3 is complete !
trans_complete_2  [2]     ! 1 = transaction_2 is complete !
trans_complete_1  [1]     ! 1 = transaction_1 is complete !
trans_complete_0  [0]     ! 1 = transaction_0 is complete !
notes                   ! (Status_SE/Clear_SE, Present State, Mask, Status/Clear)   !
notes                   ! interrupt register designed to create multiple addresses  !
end_ireg

beg_mem MEM  OFFSET 0x10  DEPTH 0x08  BIT_WIDTH 8  DEFAULT 0x00
title           ! Memory0 !
b7_0             RW  [7:0]   ! info here !
notes           ! memory notes !
end_mem

end_block
```

Figure 2 RDEF example showing a register, interrupt register and memory in an IP block

A. ***RDEF Processing***

A python parser reads the RDEF files. The script extracts the information in the RDEF files, translates it to a basic xml representation of all the data. This provides a structure that is easily manipulated and readable for debug. The script replaces block instances by the actual block information, replicating as described in the instantiations. Includes, overrides and exclusions are applied from the top level through the system. The final xml representation of the registers is converted to a binary form.

The binary representation of the registers is a tree of Device, Block, Register class that is use to generate all the design and verification files. During this phase, the scripts generate the complete register names and addresses. The binary representation is used instead of the xml representation to speed up the file generation process. The classes contain helper functions to iterate over registers/blocks at a specific level or to get the downward tree information. The complete flow is represented in Figure 3

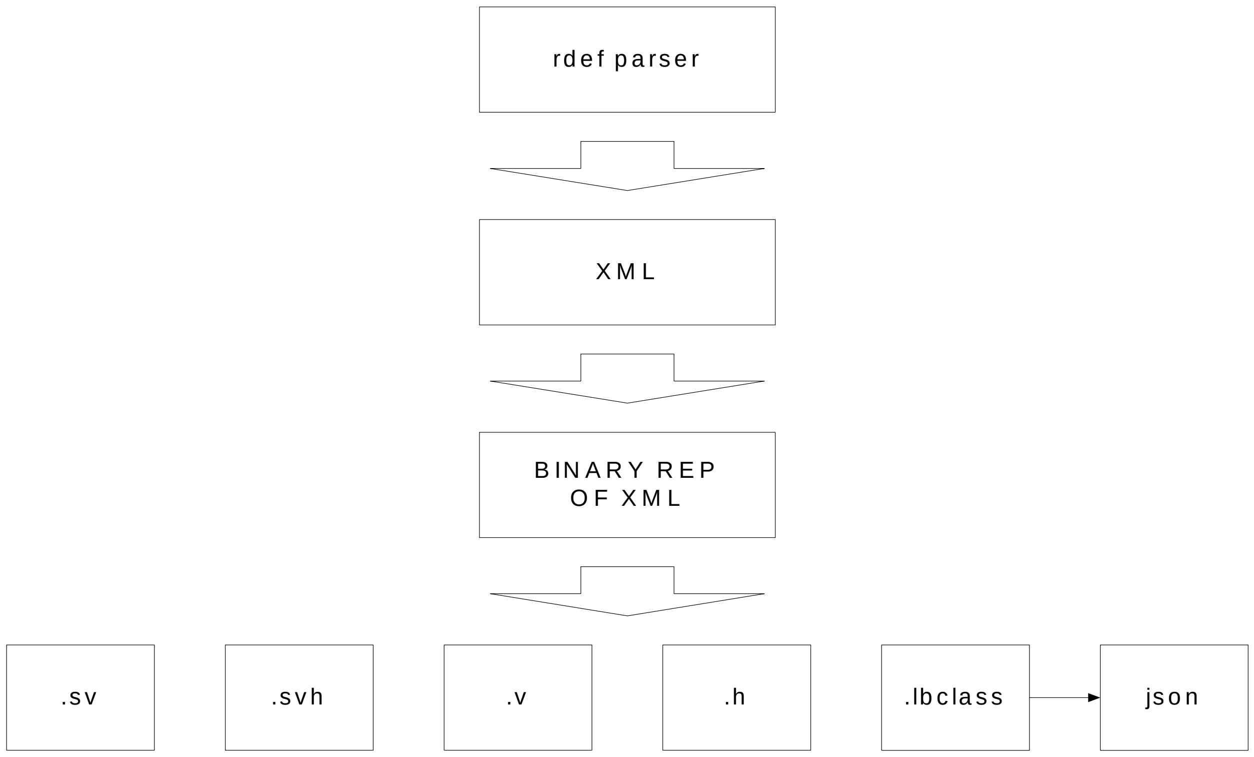


Figure 3 Rdef parsing and file generation

IV. DOCX DOCUMENTS

Hdocx allows the SoC documentation to be made up of small easily edited sections. These sections are combined at the IP level to produce large professionally presentable documents. Each IP/Device has docx templates that are created at initial creation of a design. The templates are based on a standardized company format. Header and footer data, entered in the design database, is applied to the entire docx template files. The template files, edited by the designers, contain design specific data. The names of the template files do not change. Four of the templates files are special in that they are portable for the final design document. Each IP can contribute these four sections to the final device or system docx document. The merging is illustrated in Figure 4.

A. *Document Processor*

The docx creation is separated by documents/database entries. Two document types exist, CHIP and IP. The documentation final structure is different for each type and CHIPs can contain IPs or other CHIPs. The final docx document is an aggregation of docx from the current projects, relevant section of included IPs and generated registers descriptions. The documents included is controlled in the top RDEF file of the project. The documentation is divided into seven templates that represent the major sections of a specification. For each section the scripts combine the IP and device information. The docx format is a zip file containing xml files, images and other documents. Image, reference, main text and footnotes are extracted from each specific docx and the merge in the final document. A predefined style is applied to the resulting document to ensure uniformity in the documentation presentation. The IP docx's included with the CHIP docx are defined in the top level RDEF file. The Feature Description, Feature Overview, Functional Overview, Software Requirements templates are the possible chapters included in the final docx document.

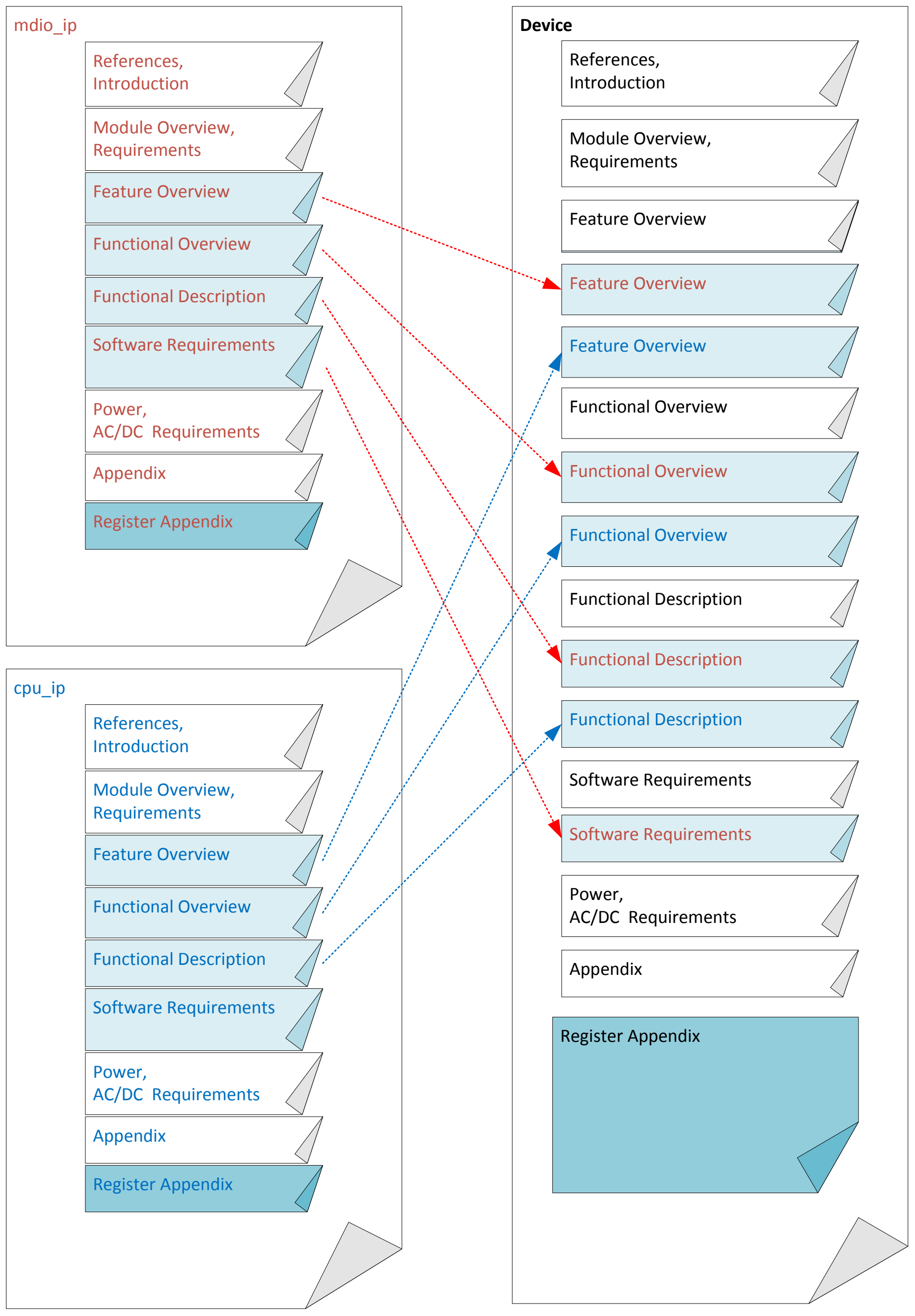


Figure 4 Documentation merge

Document generation and RDEF processing are executed on separate Linux machine using a grid engine. A parsing or document generation task is requested by the web page. The required information is stored in a container class. The class is then serialized and saved in a text file in the work directory. A task is then launched and the process executed on one a Linux machine in the grid.

V. DATABASE

The persistent data is stored in an external database. SQLAlchemy is used as an abstraction layer for the database access which allows the system to decide which database backend to use. Using a database framework makes the management of data relationship easier. All the SQL specific commands are wrapped by python classes. A python file contains the representation of the database structure. Each class is represented in a table and each variable in that class represents a field. The database contains mainly a project, user and userProject tab.

Table 1
Database structure

Database Tab	Function
Project	Project information, name, lead, etc
User	User information
UserProject	User specific project information, path to user folder, local register etc

VI. COMPUTER CLUSTER SUPPORT

The parsing of the RDEF files, generation of the output files and the docx merging are CPU resources intensive, all work done by Hdocx is performed on an array of servers using Sun Grid Engine. Required data from the database is serialized and saved on a network drive where there is access by the Sun Grid Engine worker. For testing, it is possible to deactivate the grid and run locally. After conclusion of the processing, the generated files are moved to the user project directory and ready to be used without any modification.

VII. WEB INTERFACE

Hdocx runs with client side and server side functions. The Client side runs in a web browser (Firefox, Chrome supported).

The client side web browser serves as an interface to:

- all database entries (all chip/ip designs)
- define the top level RDEF and start the rdef parsing
- receive feedback of parsing errors and expanded register map
- start the document creation, to control the final document type, name and saved location
- the container for reading the HTML register map (post parsing results)

The server side runs on a Linux machine using a RedHat install. The latest edition uses Flask [1] as a web backend and bootstrap [2] as an html framework to keep a consistent appearance. (We have used a Django web client in earlier versions of the tool.) The main Hdocx page selects or generates a new project, get the latest documentation or register definition in an html form. Each project has an owner from whom the official documentation is fetch. The document settings contain information used to populate the header and footer sections in the generated docx. The document edit page gives an overview of the current document and gives access to specific functions related to the documents content and registers. The generation of the docx is activated by the user and is perform as a background process. Once the document processing is complete a link to download it will appear. The user can leave the website and return at a later time, this will not interrupt the process. If errors occur during processing, the link provides an error log for debug.

An “add document template” button copies the default templates to the project based in the local path. The designer updates the templates using Microsoft Word and saves them with the same name. Images and external files can be included in the templates; these are merged with the rest.

The web page shown in Figure 5 makes periodical access through Ajax to the web server to check the status of the job. Users can leave the web page and come back later to check the status of the running task. Qstat is used to detect when the task is completed. Once the task is complete, the server verifies that the required files have been generated. In docx generation case, the resulting docx is copied to the static file directory so the user can download it. The web page displays a link with the job completed indication. For rdef processing, upon completion of the parsing, the web server reads back the parser result from a file that contains a serialized version of the xml structure and register class.

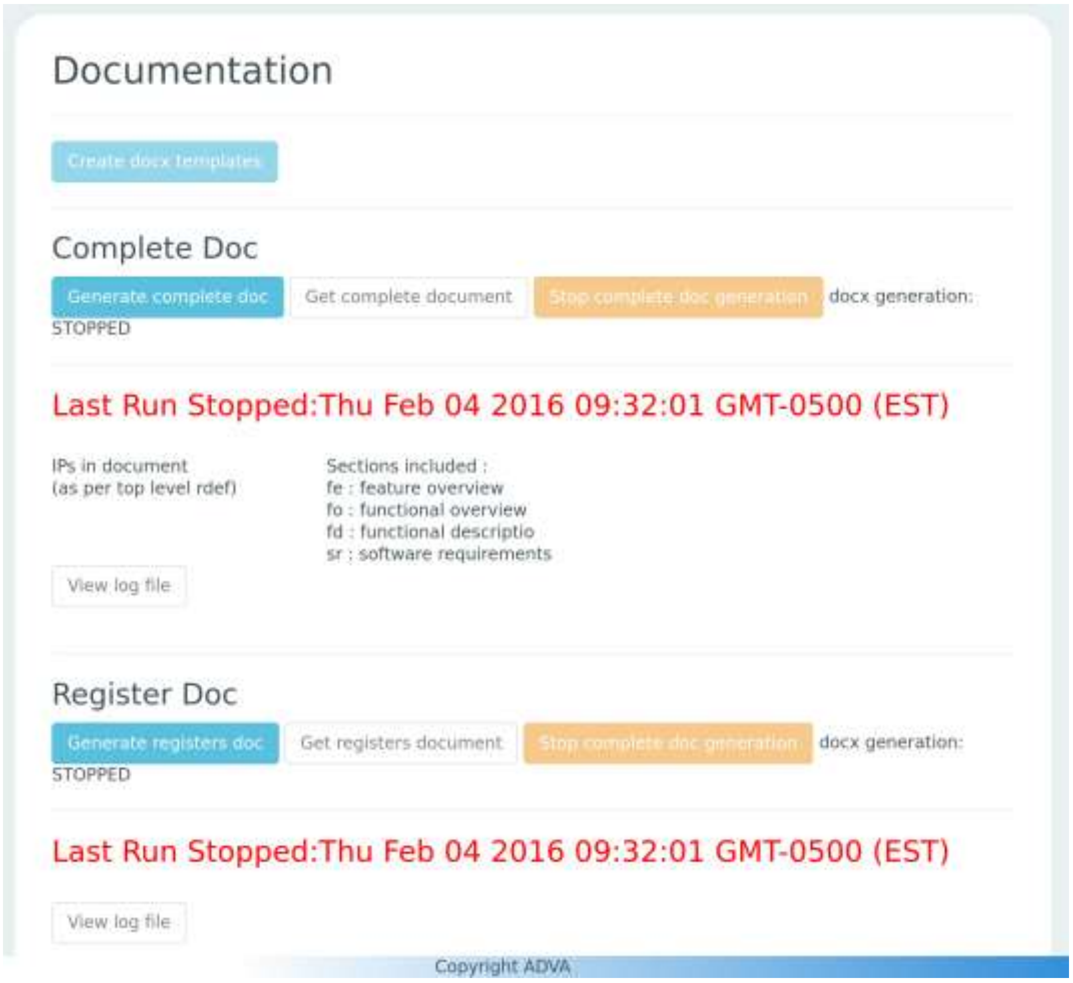


Figure 5 Web frontend

VIII. OUTPUT FILES

Output files are created for RTL design, firmware integration, System Verilog testing, lab validation and test. Using the RDEF filename syntax, the outputs allow for legacy filename compatibility. The output formats are Verilog (Figure 6) and VHDL (Figure 7) for RTL design, SystemVerilog Verification files ('plug n play' for uvm register package, including bitbash testing Figure 8 and Figure 9), custom interrupt tree test and read only register test. The complete register map including register names and field names are stored in a python pickle (serialized object) file and JSON file for firmware use. This provides seamless integration to firmware. For lab validation and/or test, Hdocx creates custom files.

```
localparam   DVCON_ID                =     8'h0;
localparam   DVCON_ID_INIT           =   16'hbabe;

localparam   DVCON_MEM               =     8'h10;
localparam   DVCON_MEM_INIT          =     16'h0;

localparam   DVCON_IRG_REG_SECE      =     8'h4;
localparam   DVCON_IRG_REG_SECE_INIT  =     16'h0;

localparam   DVCON_IRG_REG_PRES      =     8'h6;
localparam   DVCON_IRG_REG_PRES_INIT  =     16'h0;

localparam   DVCON_IRG_REG_MASK      =     8'h8;
localparam   DVCON_IRG_REG_MASK_INIT  =    16'hff;

localparam   DVCON_IRG_REG_STAT      =     8'ha;
localparam   DVCON_IRG_REG_STAT_INIT  =     16'h0;
```

Figure 6 Example of Verilog header file for address decode

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;


package param_dvcon_example_pkg_vhdl_pkg is

  constant   DVCON_ID                         : std_logic_vector(7 downto 0) := x"0";
  constant   DVCON_ID_INIT                    : std_logic_vector(15 downto 0) := x"babe";

  constant   DVCON_MEM                        : std_logic_vector(7 downto 0) := x"10";
  constant   DVCON_MEM_INIT                   : std_logic_vector(15 downto 0) := x"0";

  constant   DVCON_IRG_REG_SECE               : std_logic_vector(7 downto 0) := x"4";
  constant   DVCON_IRG_REG_SECE_INIT          : std_logic_vector(15 downto 0) := x"0";

  constant   DVCON_IRG_REG_PRES               : std_logic_vector(7 downto 0) := x"6";
  constant   DVCON_IRG_REG_PRES_INIT          : std_logic_vector(15 downto 0) := x"0";

  constant   DVCON_IRG_REG_MASK               : std_logic_vector(7 downto 0) := x"8";
  constant   DVCON_IRG_REG_MASK_INIT          : std_logic_vector(15 downto 0) := x"ff";

  constant   DVCON_IRG_REG_STAT               : std_logic_vector(7 downto 0) := x"a";
  constant   DVCON_IRG_REG_STAT_INIT          : std_logic_vector(15 downto 0) := x"0";

end package;
```

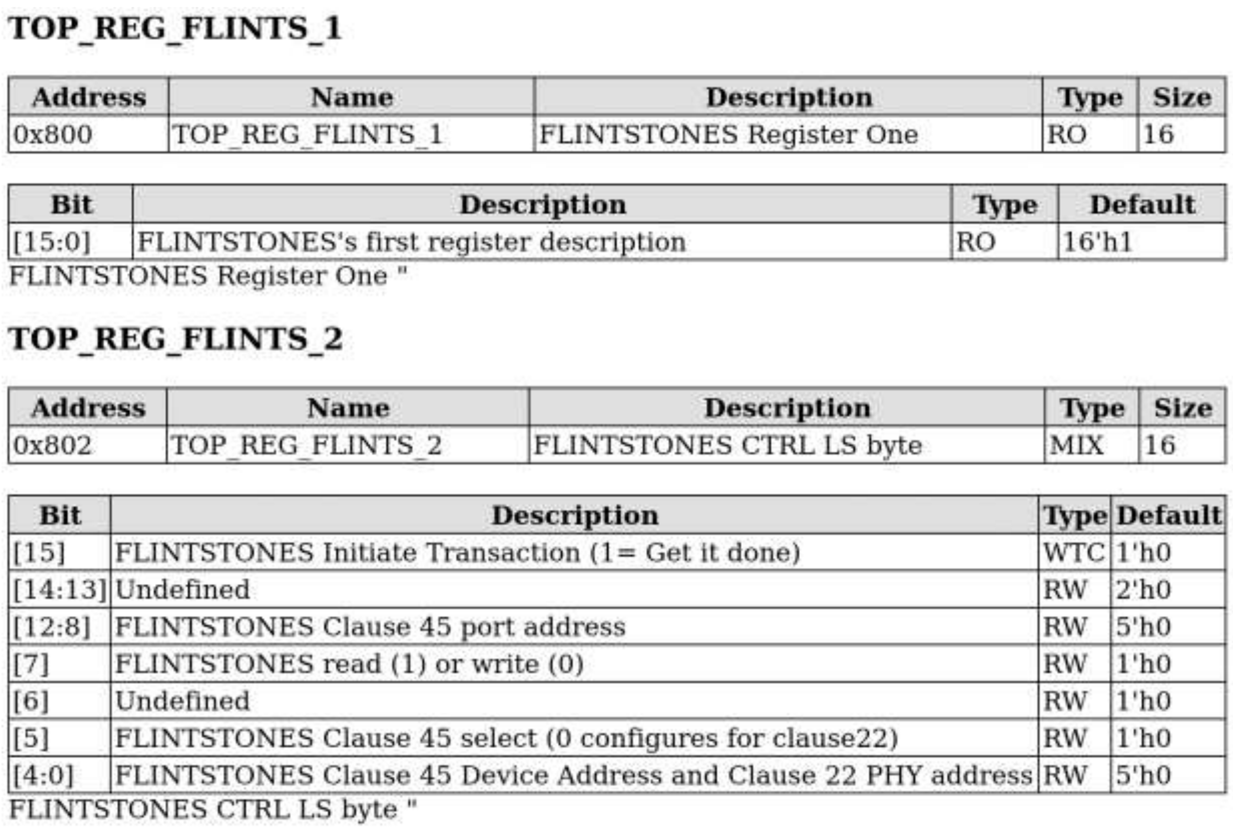
Figure 7 Example of VHDL package file for address decode

```
class dvcon_reg_tc extends reg_tc;
   `ovm_component_utils( dvcon_reg_tc );

  function new(  string name ="dvcon_reg_tc",
                 ovm_component parent =null);
     super.new( name , parent );
  endfunction

  function void build();
    super.build();
  endfunction

  function void set_mcf_mode();
    m_top_config.uvm_reg_test=UVM_REGBLK_DVCON;
  endfunction

  task start_check();
  endtask

  task stop_tc();
    final_check();  // check model for errors/warnings
    ovm_test_done.drop_objection(this);
  endtask: stop_tc

endclass : dvcon_reg_tc
```

Figure 8 Example of System Verilog test case for Bitbash Test

```
class ral_reg_DVCON_DVCON_ID extends uvm_reg;
`uvm_object_utils(ral_reg_DVCON_DVCON_ID)

  uvm_reg_field id_hi_field;
  rand uvm_reg_field id_lo_field;

  function new(string name = "DVCON_DVCON_ID");
    super.new(name, 16,build_coverage(UVM_NO_COVERAGE));
  endfunction: new

 virtual function void build();
     this.id_hi_field = uvm_reg_field::type_id::create("id_hi_field",,get_full_name());
     this.id_hi_field.configure(this, 4, 12, "RO", 0, 4'b1011, 1, 0, 0);
     this.id_lo_field = uvm_reg_field::type_id::create("id_lo_field",,get_full_name());
     this.id_lo_field.configure(this, 12, 0, "RW", 0, 12'b101010111110, 1, 0, 0);
  endfunction: build

endclass : ral_reg_DVCON_DVCON_ID
```

Figure 9 Example of UVM Register Package Bitbash Support Code



TOP_REG_FLINTS_1

Address	Name	Description	Type	Size
0x800	TOP_REG_FLINTS_1	FLINTSTONES Register One	RO	16

Bit	Description	Type	Default
[15:0]	FLINTSTONES's first register description	RO	16'h1

FLINTSTONES Register One "

TOP_REG_FLINTS_2

Address	Name	Description	Type	Size
0x802	TOP_REG_FLINTS_2	FLINTSTONES CTRL LS byte	MIX	16

Bit	Description	Type	Default
[15]	FLINTSTONES Initiate Transaction (1= Get it done)	WTC	1'h0
[14:13]	Undefined	RW	2'h0
[12:8]	FLINTSTONES Clause 45 port address	RW	5'h0
[7]	FLINTSTONES read (1) or write (0)	RW	1'h0
[6]	Undefined	RW	1'h0
[5]	FLINTSTONES Clause 45 select (0 configures for clause22)	RW	1'h0
[4:0]	FLINTSTONES Clause 45 Device Address and Clause 22 PHY address	RW	5'h0

FLINTSTONES CTRL LS byte "

Figure 10 Example of HTML Output as seen on Web Browser

IX. Retrospect

Hdocx creation occurred whilst being used for design. The development team gained knowledge that would have made the creation of the platform easier if started from scratch. It was difficult to have a tool in development while engineers are using it for designs. We ran into issues because of multi-site support with different language (English, German, Chinese, etc) and Linux systems. For locations that do not have access to Linux server, we created a Virtual Machine version of the tool. Eventually we created 'golden model' of the various syntax types to provide a baseline for feature release. The golden model continues enhancements as features increase.

X. Conclusion

Hdocx provides an easy to use interface for the following aspects of chip design.

Simplify chip design and IP integration:

- Creating reusable chip definitions for registers, memory, interrupt registers
- Grouping the definitions into reusable parts (sub-blocks, blocks, devices, systems)
- Creating design, verification and test outputs - based on the chip definition files
- Customizable memory map – outputs Verilog include files/packages and VHDL packages
- Complete integration with firmware design/testing groups

Easier design verification and validation

- UVM Register Package compatible verification registers tests (bit bash)
- Validation lab scripts outputs
- Chip interrupt support
- Interrupt tree definition built into the register definitions
- Customizable interrupt tests for verification of interrupt actions

Simplify the process of documenting the device

- All registers, register fields, memory bits, descriptions available in one file
- Creating reusable documentation based on the chip definition files and supporting design documentation
- Document templates/chapters are edited for each design IP
- Design documents are reused and combined to create the system level document
- Template based IP chapters for multiple projects
- Document changes track to all projects

Easier access to documentation and register definition

- Creating HTML formatted register maps and chip descriptions - available via url address
- HTML references to all the registers/fields and descriptions
- Hyperlinked for fast browsing
- HTML folding used for common block of registers/blocks/devices
- Read only version for non-developers

Hdocx shows that a web based client server interface provides an easy to use design environment that can be shared and updated worldwide, in real time, by one or many users. The Hdocx method has been used on FPGA designs consisting of over 30,000 registers. Hdocx has been used to create chip register maps, chip support files, verification tests, as well as a chip design documents for three large devices (10,000+ registers) and at least ten IP. Hdocx has been shared and used simultaneously by engineers in USA, Canada, Germany and China.

XI. References:

[1] http://flask.pocoo.org/
[2] http://getbootstrap.com/
http://www.ecma-international.org/publications/standards/Ecma-376.htm