从零学习从零开始练手UVM实战

通过手打并验证张强的《UVM实战》第二章是了解UVM验证方法的最佳途径。这一章节使用一个最简单的DUT开始，一步步地构建出UVM树，最后集成sequence 和sequencer，完成一个完整的基于UVM的验证平台的构建。
为了能顺利的练习这一章节的代码，可以使用带有vcs和verdi的IC开发环境，最好使用Linux版本。如果不使用Linux版本，在windows上使用QuestaSim也是可以的。

> 建议使用芯王国提供的IC虚拟机

一、VIM编辑器优化

在很多时候，Linux发行版都自带vi和vim两个软件，都可以对计算机的文本进行编辑。如果使用恰当，能极大的提高编码的效率。而且很多公司的服务器上不能进行联网，使用vim的一些插件又必须联网才能安装，因此这里经过实践，挑选了一些可以很方便手动安装的vim插件，并且结合自己的定义，能很好的适配Icer的工作需要。

1.1 对vim进行设置，家目录下的.vimrc的内容如下

```
" 不兼容vi
set nocompatible
" 语法高亮显示
syntax on
" 设置显示行号
set nu
" 设置显示当前行
set cul
" 设置相对行号，普通模式显示，编辑模式绝对行号
augroup relative_numbser
 autocmd!
 autocmd InsertEnter * :set norelativenumber
 autocmd InsertLeave * :set relativenumber
augroup END

"防止中文注释乱码
set fileencoding=utf-8
set fenc=utf-8
set fencs=utf-8,usc-bom,euc-jp,gb18030,gbk,gb2312,cp936,big - 5
set enc=utf-8
let &termencoding=&encoding

"设置字体
set guifont=DejaVu\ 14

" 设置tab4个空格
set tabstop=4
set expandtab
set shiftwidth=4

" 搜索忽略大小写
set ignorecase
" 智能缩进
set smartindent
set autoindent

set showcmd
" 颜色主题
colorscheme desert
" 退格键有效
set backspace=indent,eol,start
" 搜索高亮和历史记录
set hlsearch
set incsearch
set history=300
" 显示匹配括号
set showmatch
set autowrite
```

```
" 开启文件类型检测和插件
filetype on
filetype plugin on
filetype indent on
" set airline and colorful
set laststatus=2
set t_Co=256
" airline插件优化，不使用powerlinefont
let g:airline_detect_whitespace=0
let g:airline_powerline_fonts=0
" unicode symbols
let g:airline_left_sep = '»'
let g:airline_left_sep = '▶'
let g:airline_right_sep = '«'
let g:airline_right_sep = '◀'
let g:airline_linecolumn_prefix = '␊ '
let g:airline_linecolumn_prefix = '
 '
let g:airline_linecolumn_prefix = '¶ '
let g:airline_branch_prefix = '⎇ '
let g:airline_paste_symbol = 'ρ'
let g:airline_paste_symbol = 'Þ'
let g:airline_paste_symbol = '∥'
let g:airline_whitespace_symbol = 'Ξ'
" 自动补全弹出选项
let g:acp_completeOption = '.,w,b,k'
" 为自动补全功能加入自定义字典
set dictionary+=~/.vim/dict/sv_uvm.txt
```

这里出现了我安装的两个插件air-line和acp,这两个插件都可以从github上下载，并且解压之后扔进家目录下的.vim目录中，非常方便。air-line使你的vim有一个高大上的状态栏，acp使你的vim能够进行自动补全。

1.2 安装插件

- ☑ 首先是安装systemverilog语言插件的安装。在github下载此插件，直接将其解压后的目录全部扔进家目录下的.vim文件夹中。可以自己任选一个语言插件。我本人使用的nachumk/systemverilog.vim
  - 插件github页面 systemverilog语法高亮插件
- ☑ 然后是安装AutoComplPop(acp)，为了兼容性我选择了这个插件的2.12版本，将其下载好之后，也全部扔进家目录下的.vim文件中。然后按照上面的vimrc文件设置就可以了
  - 插件github页面 匹配弹出自动补全插件
- ☑ 接下来是安装air-line插件，为了兼容性，我选择了这个插件的0.3版本。将其下载好之后，解压之后也是全部扔进.vim文件夹即可。然后按照上面的vimrc进行配置，一般公司的服务器禁止了联网，也就让powerlinw-font的安装变得比较困难，因此采用unicode字体是折中的方案。
  - 插件github页面 airline状态栏插件

- [x] 最后是加入文件浏览窗口，方便自己打开文件，提升工作效率。同样的，为了兼容性，我选择了这个插件的6.7.5版本。同上的安装方法。此插件的工作方式是在普通模式下按：NERDtree 。
    - 插件github页面 文件浏览窗口插件

1.3 自定义字典文件让编辑更有效率

可以针对不同的文件类型进行设置不同的字典，从而达到编程效率的提升。但是这里我没有这样做，因为我在练习的时候几乎只打开makefile和sv文件。如果要这样设置的话，则要按如下的命令修改
将vimrc中的

> set dictionary+=~/.vim/dict/sv_uvm.txt

修改为 :

> autocmd FileType systemverilog set dictionary+=~/.vim/dict/sv_uvm.txt

其中黄色高亮部分应该是在air-line状态栏中显示文本名字!

字典文件是一个文本文件，按行读取，一行就是一个匹配关键字，当输入和这个字典匹配的前两个字母的时候，acp插件会自动将匹配到的关键字显示出来，此时可以通过ctrl+n和ctrl+p来选择合适的进行补全，减少编码出错也可以提升编码速度。同时也支持方向键进行选择，回车键进行确定。
如下所示，是迄今为止我自己添加的sv_uvm词典。

sv
svh
define
extern
repeat
posedge
negedge
parameter
module
endmodule
function
endfunction
class
endclass
program
endprogram
interface
endinterface
task
endtask
wire
reg
input
output
inout
logic
bit
int
always
assign
begin
end
if
for
fork
join
join_any
join_none
wait
super
virtual
extends
automatic
static
modport
clocking
initial
forever
static
push_back

pop_front
new
transaction
monitor
env
driver
include
ifndef
indef
endif
display
import
uvm_driver
uvm_pkg
uvm_macros
main_phase
build_phase
connect_phase
report_phase
starting_phase
get_starting_phase()
set_automatic_phase_objection()
uvm_phase
uvm_info
uvm_fatal
uvm_error
uvm_component
uvm_component_utils
uvm_component_param_utils
uvm_component_utils_begin
uvm_component_param_utils_begin
uvm_component_utils_end
uvm_object
uvm_object_utils
uvm_object_param_utils
uvm_object_utils_begin
uvm_object_param_utils_begin
uvm_object_utils_end
uvm_blocking_get_port
uvm_analysis_port
uvm_tlm_analysis_fifo
uvm_agent
uvm_scoreboard
uvm_config_db
uvm_transaction
uvm_env
uvm_monitor
uvm_scoreboard
uvm_field_int
uvm_field_array_int
uvm_sequencer

uvm_sequence
uvm_do
uvm_do_with
uvm_report_server
uvm_test
uvm_object_wrapper
uvm_root
uvm_sequence_item
uvm_sequence_base
uvm_sequencer_base
uvm_sequencer_param
uvm_reg_item
uvm_reg_map
uvm_mem
uvm_reg_field
uvm_reg
uvm_reg_file
uvm_reg_block
uvm_seq_item_pull_port
get_type_name()
get_full_name()
get_report_server()
get_severity_count()
req
analysis_export
blocking_export
collect
raise_objection
drop_objection
type_id
create
display
finish
stop
string
sequnence
sequnencer
seq_item_port
seq_item_export
seq_item_prod_if
start_item
finish_item
item_done
get_next_item
try_next_item
start
get
default_sequence
null
connect
is_active

```
UVM_ACTIVE
UVM_PASSIVE
UVM_LOW
UVM_MEDIUM
UVM_HIGH
UVM_FULL
UVM_DEBUG
UVM_INFO
UVM_WARNING
UVM_ERROR
UVM_FATAL
UVM_ALL_ON
pack_bytes
unpack_bytes
```

这些基本满足了《UVM实战》第二章的编辑需求，下面就愉快的开始搭建一个最基本的验证平台吧。

二、基本的验证平台

2.1 万里征途第一步

建立一个工作目录并使用git程序初始化一个仓库。下面将这个工作目录称为练习目录。

> mkdir studyUVM
> git init

按照张强老师的设计，首先手动敲入DUT的代码。在练习目录下，新建dut.v文件并使用vi程序编辑。

> vi dut.v

然后开始愉快的敲代码吧。因为之前已经设置了自动补全，你会发现vim并不是那么难用，界面也不是那么丑陋，，而且不用鼠标的编辑过程会让你不由得加快速度！
当然可以将作者的dut代码进行改进，我的做法是在端口列表的时候声明方向，这样在例化的时候就很好检查了。

dut.v

```
`ifndef dut_v
`define dut_v
module dut (
  input clk,
  input rst_n,
  input[7:0] rxd,
  input rx_dv,
  output[7:0] txd,
  output tx_en
);

reg[7:0] txd;
reg tx_en;

always @(posedge clk) begin
   if(!rst_n) begin
      txd <= 8'b0;
      tx_en <= 1'b0;
   end
   else begin
      txd <= rxd;
      tx_en <= rx_dv;
   end
end
endmodule
`endif
```

2.2 构建driver和makefile

接下来按照《UVM实战》第二章的内容，先搭建一个只有driver的验证平台。依次建立my_driver.sv和top_tb.sv.其中，my_driver在之后的一步步升级中将会慢慢改进，top_tb是顶层，负责产生时钟和复位信号，并且将dut例化，引用driver。（代码和相关的解释在UVM实战书籍中）

- ▶Tips 在所有的sv文件中，一般都会使用`ifndef 和`endif以及`define来解决重复定义和重复包含的问题。

```
`ifndef file_name
`define file_name
……code……
`endif
```

- ▶Tips 除此之外，使用UVM则我们所有的类都需要派生自UVM库中的类，因此每个文件头应该包含UVM的宏定义和引入UVM类库。

```
`include “uvm_macros.svh”
import uvm_pkg::*;
```

当把driver和top_tb都编辑完成后，请编写一份makefile脚本来启动VCS和Verdi，来检查一下driver是否真的如我们所设计的运行起来了，并且通过verdi工具来查看波形和数据以及debug。

起初，makefile可以用来管理c和c++项目的编译，使用make工具对makefile进行解读并且按照脚本进行目标生成。当然，现在make工具不仅仅能管理c和c++工程了，也可以调用任何基于shell命令行的工具来提高工作效率，对命令进行格式化并且自动化运行。

一个makefile管理的对象是其所属的文件夹以及其内部递归的文件夹。因此直接在练习目录中新建一个makefile文件。

> vi makefile

然后按照VCS和verdi工具的命令行格式，编写一个makefile。VCS有很多命令，详情可以去谷歌一下。VCS的编译选项要达成的目标：要能使用UVM库，启用SV支持以及使用64位编译，以及使用一个filelist文件来统一管理我们要编译的文件，能生成波形文件并联合verdi进行调试。Verdi的选项要达成的目标主要是：读取设计实例，读取波形文件并且可以联合调试。

VCS是一个编译型的仿真工具，将sv设计和verilog设计转换成C语言并且编译成可执行文件，运行这个可执行文件就是进行仿真的过程，所以使用VCS就是两个步骤：编译，运行。
下面是我的makefile。

Makefile

```
.PHONY:compile sim debug clean
OUTPUT = top_tb

VCS = vcs -full64 \
              -fsdb +define+FSDB \
                  -sverilog +v2k \
                  -debug_acc+all -debug_region+cell+encrypt \
                  -o ${OUTPUT} \
                  -l compile.log \
                  -ntb_opts uvm-1.2 \
                  -timescale=1ns/1ps \

SIM = ./${OUTPUT} -l sim.log

compile: clean
        ${VCS} -f filelist.f

sim:
        ${SIM}

debug:
        verdi -f filelist.f -ssf top_tb.fsdb &

clean:
        rm -rf ${OUTPUT} *.log csrc ${OUTPUT}.daidir top_tb.fsdb top_tb.fsdb.* *.key vc_hdrs.h v
```

其中，filelist.f文件是在当前目录下的一个包含所要编译sv文件和verilog文件的文本文件，现在的内容比价少，随着以后的验证平台的完善，这个文件将会越来越丰富。

filelist.f

> ./dut.v
> ./my_driver.sv
> ./top_tb.sv

在 makefile 中，使用了VCS变量、OUTPUT变量和SIM变量。VCS变量是使用vcs进行编译的常用命令选项，SIM变量的进行vcs仿真用到的选项，OUTPUT变量是指定vcs编译生成的目标，如果不指定话，vcs会默认生成一个名为simv的可执行文件，那么进行仿真就是运行simv文件了。

vcs常用编译选项

选项	参数	含义
-R		编译后立即执行仿真
-o	output filename	指定编译可执行文件名称

选项	参数	含义
-l	log filename	指定编译过程日志文件名称
-f	f filename	指定使用哪个f文件列表
-full64		启用64位编译
-fsdb	+define+FSDB	生成波形文件并且是fsdb格式
-debug_acc+all		支持调试
-debug_region+cell+encrypt		支持调试
-sverilog		启用sv支持
+v2k		对verilog2001前的语法兼容
-ntb_opts uvm-1.2		启用UVM-1.2库支持
-timescale=1ns/1ps		设置时间单位和精度
-Mupdate		只编译有更新的文件

verdi常用选项

选项	参数	含义
-f	f filename	指定使用哪个f文件列表
-ssf	fsdb filename	指定使用哪个波形文件
&		在新进程运行verdi

ϕ特别要提醒一点的时，生成波形还需要在top_tb.sv中加入几行代码
ϕ使用UVM还需要指定UVN库的路径环境变量

到这里，就可以进行编译和仿真了，尝试着在终端输入

```
make clean
make compile
```

不出意外的话，vcs将会开始编译，如果没有语法问题，那么这个简单的验证平台就能转换成可执行文件了。如果出现了错误，就根据提示使用vi编辑器去修改自己编码时出现的错误吧。

当通过编译之后，别高兴太早，仿真时可能也会莫名出现卡死等现象。通过运行make sim 命令来启动仿真。

```
make sim
```

如果按预期在屏幕上打印出了“data is dirved ”256次，那么恭喜你，可以进行下一步了。使用verdi工具来对波形进行检查吧。具体的使用细节可以通过网络学习。

```
make debug
```

此时打开verdi的图像化界面，请把dut的输入输出和时钟复位等感兴趣的信号加入波形窗口，查看数据输入dut和经过dut输出之后的变化。
至此，已经完成了最简单只有driver的验证平台的搭建，你可以使用 git 工具将这些文件加入仓库并提交作为快照。

```
git add dut.v my_driver.sv top_tb.sv filelist.f makefile
git commit -m “first commit with a simple dut and only my_driver”
```

2.3 继续code，慢慢了解，逐步完善

跟着《uvm实战》继续编码，一步一步完善你的平台。一步一步将UVM框架搭建出来，随着代码的增加，项目文件的增多，要及时更新filelist.f并且注意先后顺序，即使更新makefile以便适用新的需求。在每一个重要的时间节点，别忘了使用git工具将你的工作成果和学习心得记录下来。你甚至可以新建一个markdown来图解你的平台，自己花时间做一遍比看书强太多了。

2.4 补充说明

∮了解uvm1.1d和uvm1.2之间的差异.
在书中p50有注释，使用uvm1.2的时候应该格外注意，请通过搜索来将这部分知识补齐吧。

有的时候会有出现一些小问题，那就尝试着去解决这个问题，解决之后要想一下究竟是哪里逻辑出现了错误。我在做这一个小练习的时候就是明明自己是按书籍上的代码敲的代码，却运行不起来，不由得怀疑人生，甚至一整天都在冥思苦想。到最后却发现是自己没注意或者是自己自作聪明修改的地方出了问题。还有就是有些时候编码的规整是很重要，不论是做软件还是硬件，这都是一个要训练的过程。

另外，工具的重要性不言而喻，特别是自动补全这个功能，如果预先没有设置vim,那么编码过程也会痛苦不堪，许多重复的代码敲的手疼。当然这也是一个训练的过程，如果时间够多，可以完全手敲，体会验证平台一步一步完善的思路和过程。如果想要快速刷完这一个练习，并且不使用copy的方法，最快的方案就是调教你的vim，让它智能的帮助你。在编码的过程中，时不时更新你的vim词典。

2023.05.18 孺子牛大厦
ZhaoPeng @ BOSC