Verilator + gtkwave 快速上手教程 v1.8

0. preview

0.0 软件简介

Verilator 可以将 Verilog 集成到 C++程序中，实现综合仿真。
The Verilator package converts Verilog and SystemVerilog hardware description language (HDL) designs into a C++ or SystemC model that after compiling can be executed. Verilator is not a traditional simulator, but a compiler.

Gtkwave 可以使波形文件可视化。

0.1 工程文件构成

0.2 动手前看看

$(variable)表示命名为 variable 的字符串变量

1. 编辑 verilog 顶层文件$(top).v：实现模块内部逻辑，向外部提供 IO 信号端口。
2. 编辑应用程序代码:在测试文件$(test).cpp 中包含头文件 #include "V$(top).h"，对类 V$(top) 进行实例化并访问。

V$(top) 这个类将可读写的 IO 信号作为成员变量，提供给应用程序。顶层模块中的子模块为内部构件，一般不被应用程序代码触及。

若顶层模块内提供了时钟信号，则在应用程序可通过改变类的端口电平模拟时钟周期变化。例如顶层文件 Top.v，提供时钟信号端口 clock，应用程序经过 #include “VTop.h” 引入，通过以下过程模拟时钟变化：

```
VTop* top_ptr = new VTop;
...
Top_ptr->clock = 0;
Top_ptr->eval();
Top_ptr->clock = 1;
Top_ptr->eval(); //一个时钟周期后评估输出
...
```

0.3 我的环境

虚拟机软件：VMware Workstation 15 Pro 或 Oracle VM VirtualBox
操作系统：Linux ubuntu 20.04

1. 软件安装

我们提供下面两种安装方式，不建议使用命令 apt install verilator 来安装，因为该命令安装的 verilator 版本比较旧，我们推荐用新版本的 verilator。

1.1 Verilator

1.1.1 方式一 自动化脚本安装（推荐）

从”一生一芯”网站上下载 verilator_installer.tar.gz，在 Ubuntu 中解压，进入解压后的目录，使用下面的命令安装

```
./install_verilator.sh
```

该安装脚本将自动安装 verilator_4_204_amd64.deb 和所需依赖包。

1.1.2 方式二 编译安装 Verilator

```
# Prerequisites:
```

```
sudo apt-get install git perl python3 make autoconf g++ flex
bison ccache libgoogle-perftools-dev numactl perl-doc libfl2
libfl-dev zlibc zlib1g zlib1g-dev

git clone https://github.com/verilator/verilator # Only first
time
## Note the URL above is not a page you can see with a browser,
it's for git only
# Every time you need to build:
unsetenv VERILATOR_ROOT # For csh; ignore error if on bash
unset VERILATOR_ROOT # For bash
cd verilator
git pull # Make sure git repository is up-to-date
git checkout v4.204

autoconf # Create ./configure script
./configure --prefix=/usr # Configure and create Makefile
make # Build Verilator itself
sudo make install
```

具体请参考 https://verilator.org/guide/latest/install.html

1.2 Gtkwave

```
sudo apt-get install gtkwave
#gtkwave --version
```

可查看版本

Gtk-Message: 16:44:00.457: Failed to load module "canberra-gtk-module"
GTKWave Analyzer v3.3.103 (w)1999-2019 BSI

2. 获取代码

按照下面的命令下载代码，并对 git 做相应的配置

```
# 克隆 github 上的 oscpu-framework 代码
git clone --recursive -b 2021 https://github.com/OSCPU/oscpu-
framework.git oscpu
# 使用你的编号和姓名拼音代替双引号中内容
git config --global user.name "2021000001-Zhang San"
# 使用你的邮箱代替双引号中内容
git config --global user.email "zhangsan@foo.com"
# 修改你喜欢的编辑器为 git 编辑器
git config --global core.editor vim
# 让 Git 显示颜色
git config --global color.ui true
```

在 oscpu 目录中的 myinfo.txt 文件里填写自己的 ID 和姓名。接下来就可以开始编译和仿真了。

3.例程

3.1 4 位计数器

examples/counter 目录下存放了 4 位计数器的例程源码。包含 top.v 和 test.cpp 两个文件。

3.1.1 编译+仿真

执行下面的命令进行编译和仿真。

```
./build.sh -e counter -b -s
```

如果前面的步骤成功，执行后将看到下面的输出：

这是 build.sh 和 c++测试程序输出的 log。在 counter 例程的 c++测试程序中，要求我们输入一个周期数来进行 counter 的仿真，这里以 20 为例。输入 20 后程序结束运行，在 examples/counter/build/目录中生成波形文件 vlt_dump.vcd。

build.sh 的参数可以使用命令"./build.sh -h"查看：
-e 指定一个例程作为工程目录，如果不指定，将使用"cpu"目录作为工程目录
-b 编译工程，编译后会在工程目录下生成"build"(difftest)或"build_test"子目录，里面存放编译后生成的文件
-t 指定 verilog 顶层文件名，如果不指定，将使用"top.v" 或"SimTop.v"(difftest)作为顶层文件名，该参数在接入 difftest 时无效
-s 运行仿真程序，即"build/emu"程序，运行时工作目录为"build"(difftest)或"build_test"子目录
-a 传入仿真程序的参数，比如：-a "1 2 3 ......"，多个参数需要使用双引号
-f 传入 c++编译器的参数，比如：-f "-DGLOBAL_DEFINE=1 -ggdb3"，多个参数需要使用双引号，该参数在接入 difftest 时无效
-l 传入 c++链接器的参数，比如：-l "-ldl -lm"，多个参数需要使用双引号，该参数在接入 difftest 时无效
-g 使用 gdb 调试仿真程序
-w 使用 gtkwave 打开工作目录下修改时间最新的.vcd 波形文件
-c 删除工程目录下编译生成的"build"文件夹
-d 接入香山 difftest 框架
-m 传入 difftest 框架 makefile 的参数，比如：-m "EMU_TRACE=1 EMU_THREADS=4"，多个参数需要使用双引号

目录树

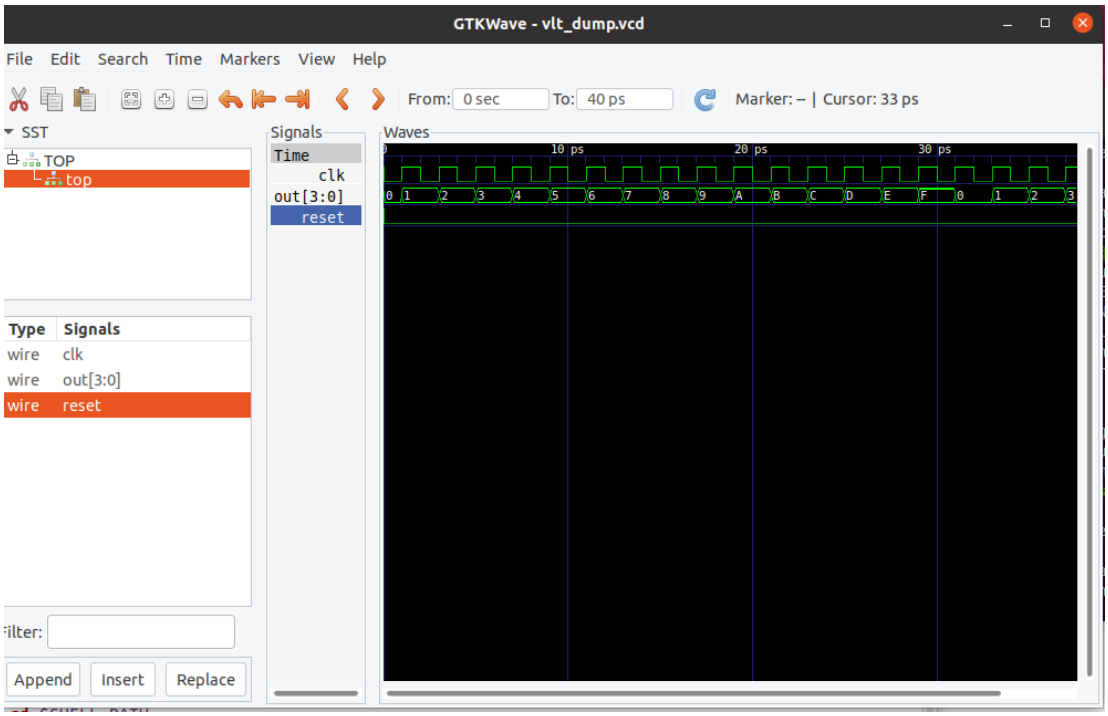
```
oscpu
├─── NEMU  //用于 difftest 参考的 cpu 模拟器
│      ├─────......
├─── README.md
├─── bin   //存放 riscv64 机器码文件的目录
│      ├─── inst.bin
│      └─── inst_diff.bin
├─── build.sh  //编译脚本
├─── difftest  //香山 difftest 框架目录
│      ├─────......
├─── projects  //项目代码目录
│      ├─── counter
│      │      ├─── csrc
│      │      │      └─── main.cpp
│      │      └─── vsrc
│      │          └─── top.v
│      ├─── cpu
│      │      ├─── csrc
│      │      │      └─── main.cpp
│      │      └─── vsrc
│      │          ├─── defines.v
│      │          ├─── exe_stage.v
│      │          ├─── id_stage.v
│      │          ├─── if_stage.v
│      │          ├─── inst.bin
│      │          ├─── regfile.v
│      │          └─── rvcpu.v
│      ├─── cpu_diff
│      │      ├─── csrc
│      │      └─── vsrc
│      │          ├─── SimTop.v
│      │          ├─── defines.v
│      │          ├─── exe_stage.v
│      │          ├─── id_stage.v
│      │          ├─── if_stage.v
│      │          └─── regfile.v
│      ├─── dpi-c
│      │      ├─── csrc
│      │      │      └─── main.cpp
│      │      └─── vsrc
│      │          └─── top.v
│      └─── finish
│          ├─── csrc
│          │      └─── main.cpp
```

```
|           └──── vsrc
|                 └──── top.v
└──── myinfo.txt  //存放个人信息的文件
```

3.1.2 查看波形

运行下面的命令，脚本将使用 gtkwave 打开波形文件。可以看到在每个 clk 的上升沿时，out 在[0,15]之间循环递增，符合设定的 4 位计数器的程序行为。

```
./build.sh -e counter -w
```



3.2 cpu

projects/cpu 目录下存放了单周期 risc-v cpu 框架源码，源码实现了 RV64I 指令"addi"。

3.2.1 编译+仿真+查看波形

使用下面命令实现对该工程的编译+仿真+查看波形。

```
./build.sh -b -t rvcpu.v -s -w
```

输入"inst.bin"+回车后程序结束运行，并打开输出的波形文件。

3.3 cpu_diff

projects/cpu_diff 目录下存放了接入香山 difftest 的单周期 risc-v cpu 框架源码，源码实现了 RV64I 指令"addi"。
如果内存小于 8G，编译该工程之前，需要修改 NEMU 和 difftest 的 ram 大小。
文件：NEMU/include/memory/paddr.h

```
//#ifdef _SHARE
//    #define PMEM_SIZE (8 * 1024 * 1024 * 1024UL)
//#else
    #define PMEM_SIZE (256 * 1024 * 1024UL)
//#endif
```

文件：difftest/src/test/csrc/common/ram.h

```
#define EMU_RAM_SIZE (256 * 1024 * 1024UL)
//#define EMU_RAM_SIZE (8 * 1024 * 1024 * 1024UL)
```

将 NEMU 编译为动态库。

```
sudo apt-get install libreadline-dev libsdl2-dev
cd NEMU
make ISA=riscv64 SHARE=1
```

3.2.1 编译+仿真+查看波形

使用下面命令实现对该工程的编译+仿真+查看波形。

```
./build.sh -e cpu_diff -d -b -s -a "-i inst_diff.bin --dump-wave
-b 0" -m "EMU_TRACE=1" -w
```

3.4 回归测试

在实现了能够运行所有 cpu-tests 测试用例的指令后，可以通过以下命令实现对 cpu 的一键回归测试。该命令会将"bin"目录下的所有.bin 文件作为参数来调用接入了 difftest 的 emu 仿真程序。

```
./build.sh -e cpu_dif -b -r
```

通过测试的用例，将打印"PASS"。测试失败的用例，打印" FAIL"并生成对应的 log 文件，可以查看 log 文件来调试，也可以自己开启波形输出来调试。