

文档版本: Ver1.0  
最后修改日期: 2015-05-30  
修改人: William



## E 课网 - UVM 实战培训



[www.eecourse.com](http://www.eecourse.com)

[klin@eecourse.com](mailto:klin@eecourse.com)

## SVV 实验 - Lab01\_Guide

## 实验简介:

通过本次实验练习，你将熟悉 QuestaSim 工具的基本使用方法，同时来理解构成一个测试平台（以下称为 Testbench）最基本的组件 – interface, program body 和 top-level 集成。在该试验中，你也会学习在 Testbench 中使用一些基本的数据类型。

## 实验目的:

- ★ 熟悉 QuestaSim EDA Tool 的操作方法，特别是在 Terminal 中使用命令行的方式使用 QuestaSim;
- ★ 熟悉使用 Makefile 脚本调用 Quesasim Tool 进行仿真;
- ★ 对一个典型的基于 SystemVerilog 的 Testbench 有一个大致的感受;
- ★ 认真阅读 DUT 手册，熟悉课程所要验证的设计模块的结构和功能。

## 实验准备:

- 使用 VNC 登录服务器，进入你的工作目录中，

```
cd /qixin/proj_users/<your_work_directory>
```

通过 SVN 将实验代码 checkout 出来，在 terminal 中键入：

```
svn co file:///qixin/project/svn_depot/verification
```

等待命令执行完毕，会在当前目录下出现一个 verification 目录，然后进入

```
cd verification/svv/lab01
```

中，该目录中包含了本次实验所需要的相关代码和文档。

如果已经 checkout 过的，该步骤可跳过。

- 关于服务器的登录以及 SVN 的使用方法请参照文档《[网络远程登录服务器登陆方法及数据库版本管理 - VNC SVN.pdf](#)》
- 请确认你已经获得本次实验的 DUT 参考手册《[E 课网 UVM 实战培训 SVV 实验 DUT 手册.pdf](#)》
- 如果在上述准备工作中遇到任何困难，请及时与讲师联系解决。

## 实验步骤:

### 1. 熟悉 QuestaSim DEA Toll

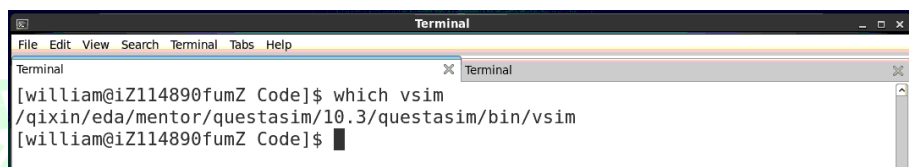
#### ①. QuestaSim 简介:

QuestaSim 是由 Mentor Graphics 公司推出的一款功能强大的仿真工具，支持 SystemC, Verilog, SystemVerilog 以及 VHDL 等硬件描述语言。它是 Modelsim 的加强版。QuestaSim 支持多种验证特性，比如 coverage databases, coverage driven verification, assertions, SystemVerilog constrained-functionality 等。

#### ②. 确认 Linux 系统环境中的 QuestaSim 可用:

在 Terminal 中键入如下命令

```
which vsim
```



看看是否返回一条如上图的信息。

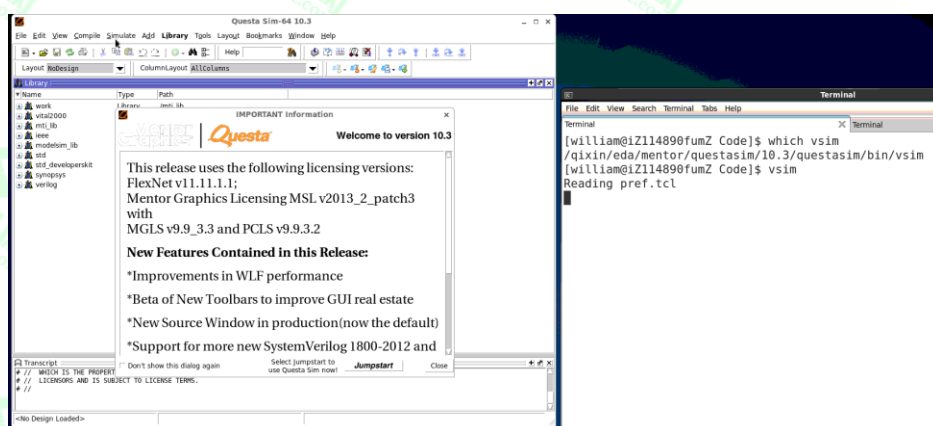
如果有，则说明系统的 QuestaSim 可用，并且可以看到其安装路径。

③. 打开 QuestaSim GUI，熟悉界面：

在 Terminal 中键入

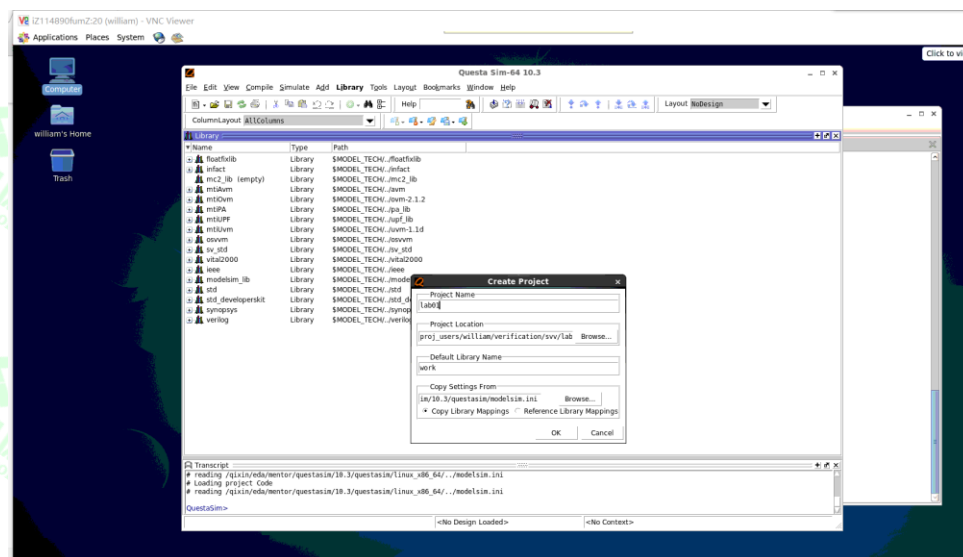
vsim

即可打开 QuestaSim 的 GUI 界面



④. 创建 project 和 library 文件夹：

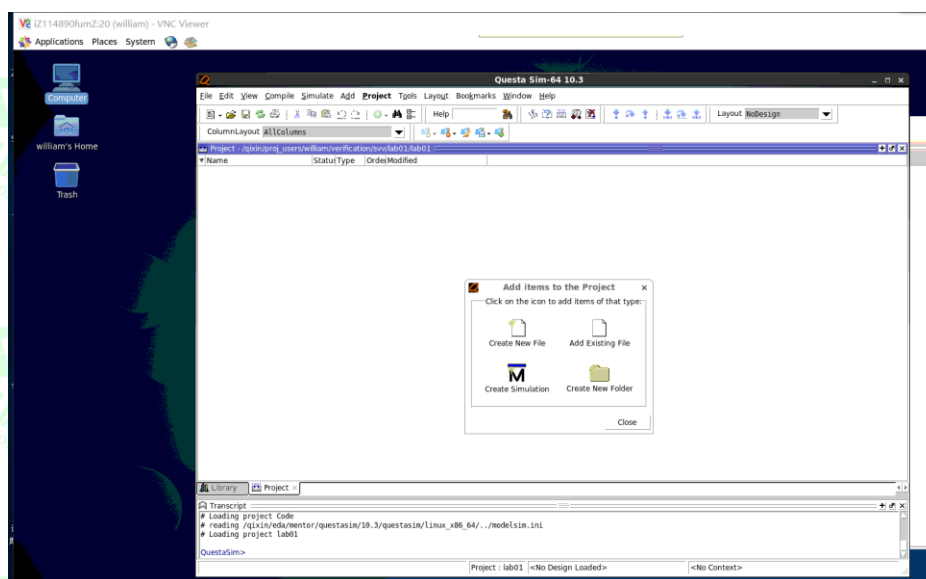
点击菜单栏上的“File”->“New”->“Project”



对于每一个项目，需要建立一个 Project，“Project Name”为新建 Project 的名称，这里我们命名为 lab01；“Project Location”为该 Project 所在的目录，可以看到即为当前目录；“Default Library Name”为仿真所用的相关文件，其中包括 QuestaSim 编译之后产生的中间文件以及用户存放的相关库文件等等，这里我们用默认的。

⑤. 向 project 中添加代码文件：

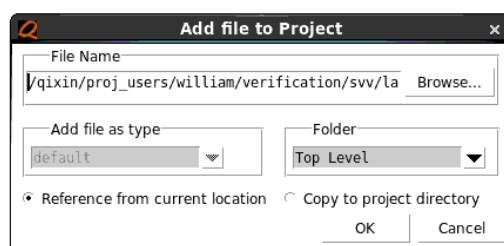
点击上一步骤中的“OK”之后，会显示



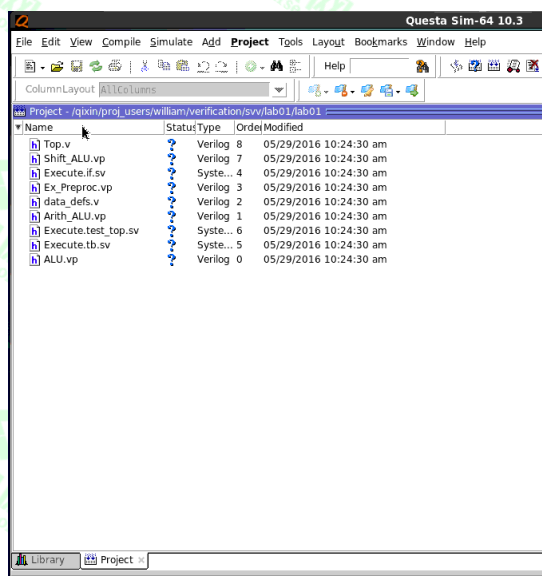
意思是用什么样的方式来加入我们的代码。

“Create New File”指的是如果我们没有代码文件，这里可以点击该图标创建新的代码文件，新创建的文件是空的，需要我们自己写代码进去；

“Add Existing File”指的是向 project 中加入已有的代码文件，这里我们的实验代码已经提供了，所以直接点击这个图标。

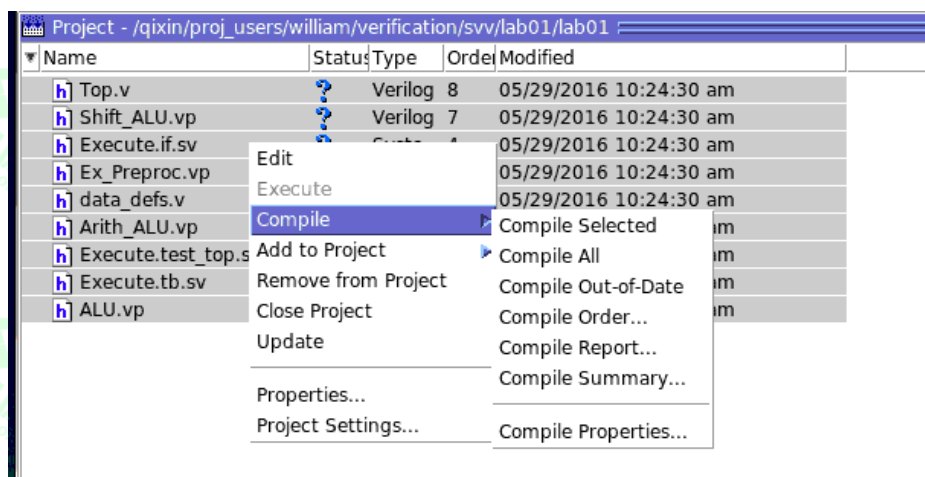


点击 Browse，进入 code 文件夹，选中所有以“.v”和“.vp”结尾的文件，然后点击 OK。添加完毕之后，可以看到一共有 9 个文件。



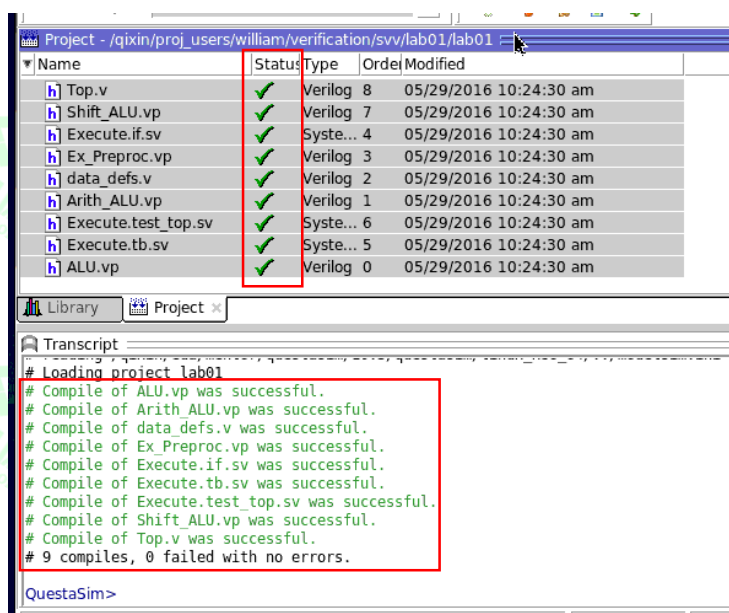
#### ⑥. 编译文件：





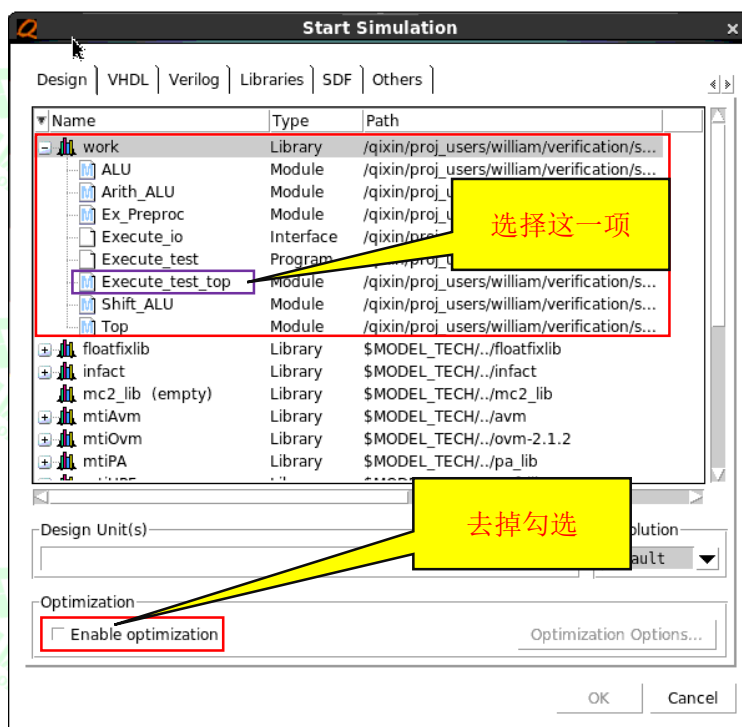
用鼠标框选所有文件，然后右键出现菜单，点击“compile”->“compile All”。

如果编译通过，可以看到 status 全部由原来的“？”变为“√”。并且在下方的报告窗口中可以看到编译成功的字样。如果编译不成功，有问题的文件的 status 为“×”，并且下方的报告窗口中也会提示出更加详细的错误信息。在 Lab01 中的代码都是正确的，所以正常情况下应该全部通过。

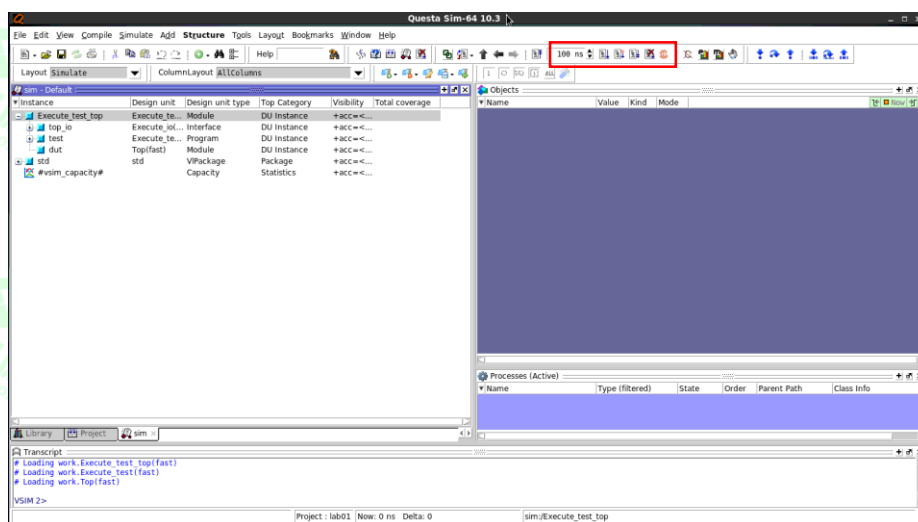


#### ⑦. 运行仿真：

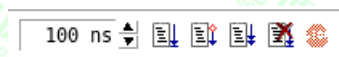
在编译通过之后，就可以运行仿真了。点击菜单栏上的“simulate”->“start simulation”，出现 Start Simulation 对话框，点击“work”前面的“+”进行展开。work 即为库目录，就是在我们创建 project 时定义的，如果当时不是使用默认值“work”，而是使用自定义的名称，那么也会出现在该对话框中，找到相应的名称即可。在“work”的展开项目中，可以看到刚才编译成功的 9 个文件。此时，我们要选择最顶层的文件“Execute\_test\_top”。在“Optimization”一栏中，去掉前面的勾选，如果勾选，QuestaSim 会在仿真时对设计进行优化，这里我们不需要它进行优化。我们在仿真时，总是要选择最顶层的模块进行仿真，这一点需要注意。



设置好之后，点击 OK。此时，进入仿真界面。



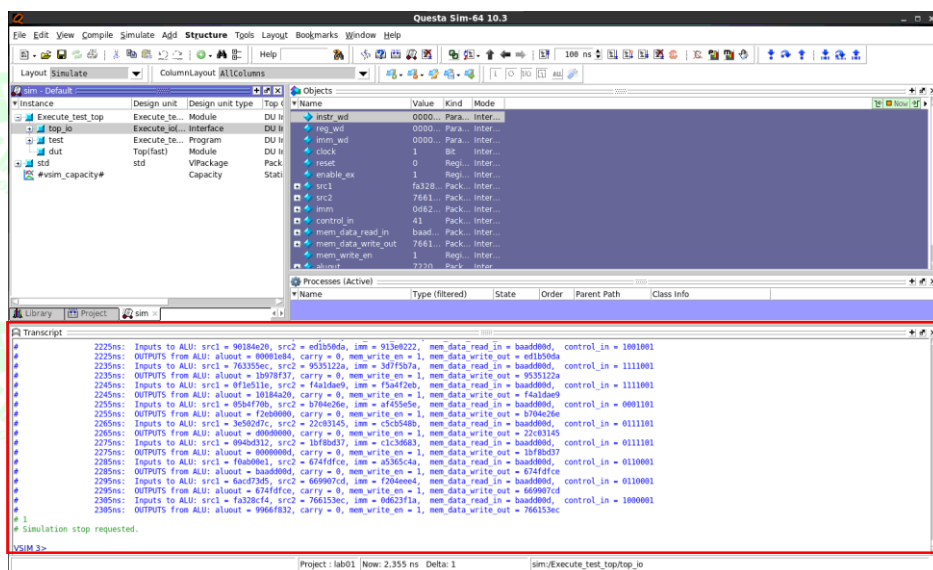
这是仿真已经准备好了，需要点击运行开始仿真。



5 个图标从左到右依次为“Run”“ContinueRun”“Run-All”“Break”“Stop”  
这里我们点击第三个“Run-All”，意思是让仿真一直运行直到结束。

#### ⑧. 查看结果

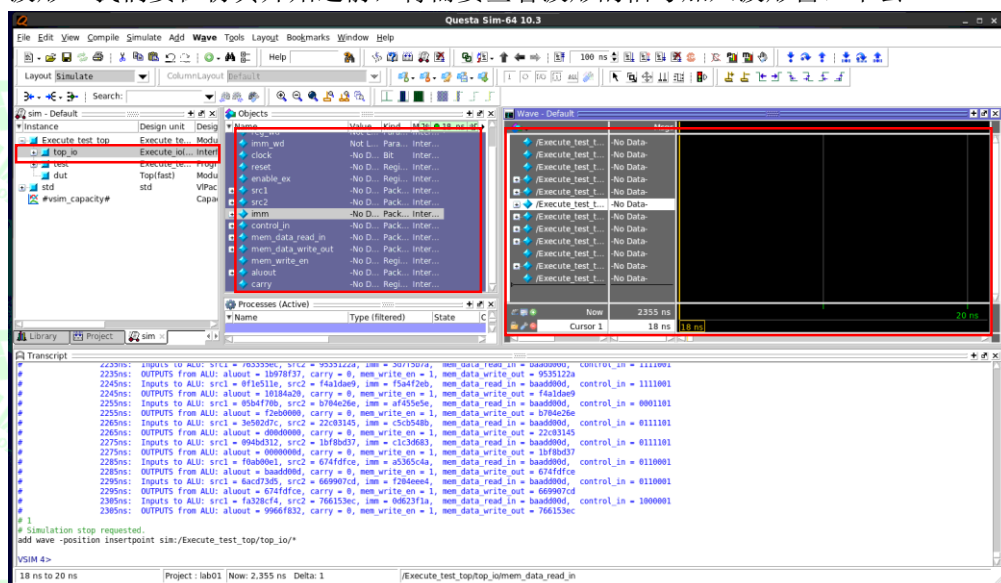
当仿真完成之后，弹出对话框，我们点击“No”，此时就不会退出 QuestaSim，我们就可以查看仿真的结果了。



可以在 Transcript 中看到仿真打印的相关信息。

### ⑨. 查看波形：

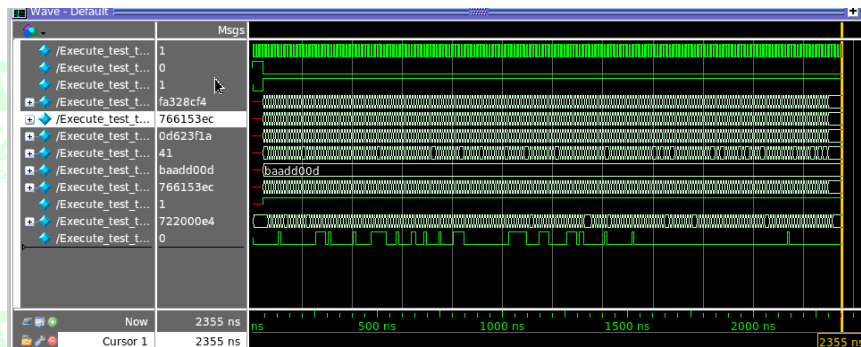
在代码中并没有加入 dump 波形的语句，所以，当上面的仿真结束之后，无法查看波形。我们要在仿真开始之前，将需要查看波形的信号加入波形窗口中去。



在“Instance”窗口中点击“top\_io”之后，会在 Objects 窗口中将信号全部选中，右键，“Add Wave”，就会在 Wave 窗口中出现这些信号，处于该窗口中的信号会在仿真时 dump 波形。

之后，我们要重新在仿真一次，点击菜单栏的“Simulate” -> “Restart...”在弹出的窗口点击“OK”。此时就可以再次进行仿真了。点击“Run -All”。





仿真结束之后就可以看到波形了。

## 2. 使用命令行来进行仿真

在实际的工程中，一般很少用到 QuestaSim 的 GUI 界面，除非要通过波形来 Debug，除此之外都是在命令行模式下进行的，那么，如何通过命令行来实现上面讲到的仿真过程呢？

进入 Lab01/Code 目录下，在 Termin 中使用

```
rm -rf work mti_lib transcript modelsim.ini
```

命令来清理掉刚才仿真所产生的中间文件。

下面，我将在 Terminal 中进行仿真：

### ① 创建 work Library:

在 Terminal 中不需要创建 Project 目录，但是需要创建仿真所需的 Library。创建的方法是

```
vlib work
```

执行这一条命令之后，就创建了 work Library，可以在当前目录下找到这个目录。

```
[william@iZ114890fumZ Code]$ vlib work
[william@iZ114890fumZ Code]$ ls
ALU.vp      Execute.if.sv  Ex_Preproc.vp  Top.v
Arith_ALU.vp Execute.tb.sv  modelsim.ini   work
data_defs.v Execute.test_top.sv Shift_ALU.vp
[william@iZ114890fumZ Code]$
```

这步工作就相当于之前的用图形界面创建“Project 和 work Library”的工作。

### ② 编译代码文件:

在之前的图形界面中，我们需要先将代码文件加入到 Project 中，然后在进行编译。我们可以在 Terminal 中使用命令一步来完成，在 Terminal 中键入（一次性输入）：

```
vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp
Shift_ALU.vp ALU.vp Top.v Execute.if.sv Execute.tb.sv
Execute.test_top.sv
```

vlog 是 QuestaSim 的编译命令，-l comp.log 表示将编译的相关信息写入到 comp.log 文件中，-sv 表示使用 SystemVerilog 的语法编译文件，后面的是 9 个代码文件名称。



```
[william@iZ114890fumZ Code]$ vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp ALU.vp Top.v Execute.if.v Execute.tb.v Execute.test_top.v
QuestaSim-64 vlog 10.3 Compiler 2014.01 Jan 6 2014
Start time: 13:08:27 on May 31,2016
vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp ALU.vp Top.v Execute.if.v Execute.tb.v Execute.test_top.v
-- Compiling module Ex_Preproc
-- Compiling module Arith_ALU
-- Compiling module Shift_ALU
-- Compiling module ALU
-- Compiling module Top
-- Compiling interface Execute_io
-- Compiling program Execute_test
-- Compiling module Execute_test_top

Top level modules:
    Execute test top
End time: 13:08:27 on May 31,2016
Errors: 0, Warnings: 0
[william@iZ114890fumZ Code]$
```

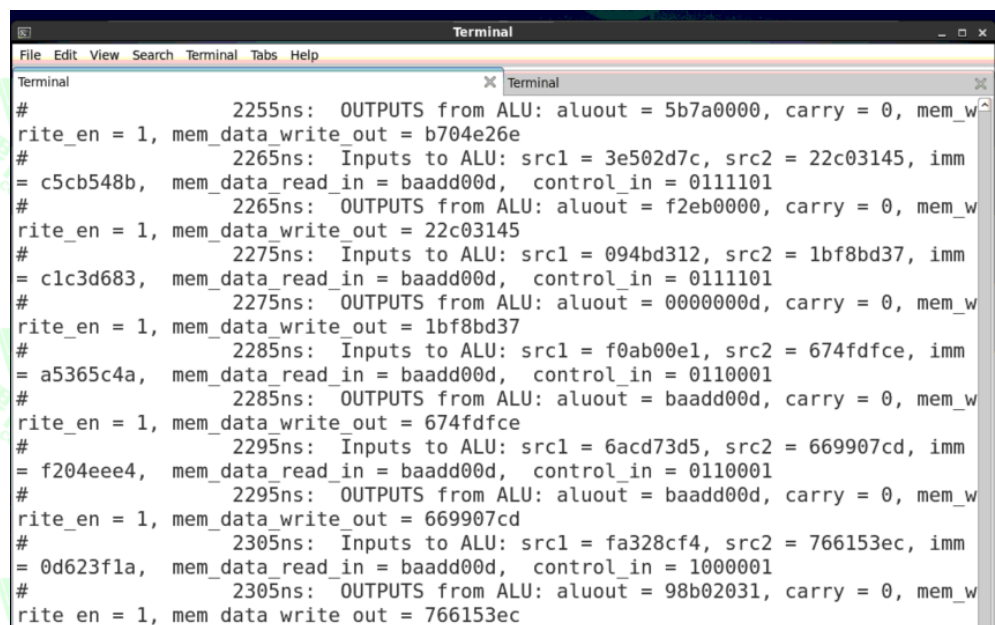
编译完毕之后，可以看到 Top level modules 是“Execute\_test\_top”表示最顶层的模块。到此就实现了文件的编译工作。

### ③ 运行仿真：

我们依然在 Terminal 中运行仿真。执行：

```
vsim -l sim.log -c -novopt Execute_test_top -do "run -all"
```

其中，-l sim.log 表示将仿真相关的打印信息写入 sim.log 文件中；-c 表示不启动 QuestaSim 的 GUI 界面，如果不加，则会启动 GUI 界面；-novopt 指的是不进行优化，这与当时在图形界面启动仿真时，去掉“enable Optimization”的效果是一样的；Execute\_test\_top 指的是对哪个模块进行仿真，这里是最顶层的模块，这与当时在图形界面指定“work”展开下面的“Execute\_test\_top”是一个意思；-do“run -all”指的就是你点击“Run -All”按钮开始仿真。



```

# 2255ns: OUTPUTS from ALU: aluout = 5b7a0000, carry = 0, mem_w
rite_en = 1, mem_data_write_out = b704e26e
# 2265ns: Inputs to ALU: src1 = 3e502d7c, src2 = 22c03145, imm
= c5cb548b, mem_data_read_in = baadd00d, control_in = 0111101
# 2265ns: OUTPUTS from ALU: aluout = f2eb0000, carry = 0, mem_w
rite_en = 1, mem_data_write_out = 22c03145
# 2275ns: Inputs to ALU: src1 = 094bd312, src2 = 1bf8bd37, imm
= clc3d683, mem_data_read_in = baadd00d, control_in = 0111101
# 2275ns: OUTPUTS from ALU: aluout = 0000000d, carry = 0, mem_w
rite_en = 1, mem_data_write_out = 1bf8bd37
# 2285ns: Inputs to ALU: src1 = f0ab00e1, src2 = 674fdfce, imm
= a5365c4a, mem_data_read_in = baadd00d, control_in = 0110001
# 2285ns: OUTPUTS from ALU: aluout = baadd00d, carry = 0, mem_w
rite_en = 1, mem_data_write_out = 674fdfce
# 2295ns: Inputs to ALU: src1 = 6acd73d5, src2 = 669907cd, imm
= f204eee4, mem_data_read_in = baadd00d, control_in = 0110001
# 2295ns: OUTPUTS from ALU: aluout = baadd00d, carry = 0, mem_w
rite_en = 1, mem_data_write_out = 669907cd
# 2305ns: Inputs to ALU: src1 = fa328cf4, src2 = 766153ec, imm
= 0d623f1a, mem_data_read_in = baadd00d, control_in = 1000001
# 2305ns: OUTPUTS from ALU: aluout = 98b02031, carry = 0, mem_w
rite_en = 1, mem_data_write_out = 766153ec

```

可以在 Terminal 中看到结果。

### ④ Debug 文件

在执行 vlog 和 vsim 命令时，分别使用了 -l comp.log 和 -l sim.log。命令执行完毕之后可在当前目录下产生这两个文件，用于 Debug。

可以看到，简单的三步就可以完成仿真过程。

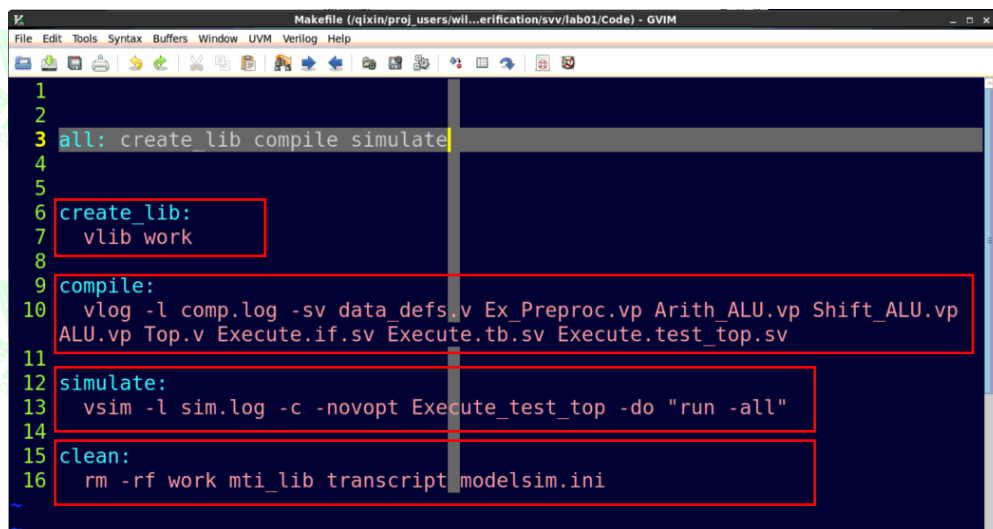
### 3. 使用 Makefile 脚本使得命令自动执行

虽然使用命令行可以很方便的完成仿真任务，但是在实际工程中，文件量巨大，每次仿真要敲这么多的命令实在是令人崩溃，我们可以将这些语句写入脚本中，在需要的时候直接调用脚本执行命令即可，下面介绍使用 Makefile 脚本来实现仿真。

#### ① 使用 gvim 来创建 Makefile 文件

gvim Makefile

#### ② 编辑该文件，将命令写入文件中



```
1
2
3 all: create lib compile simulate
4
5
6 create_lib:
7     vlib work
8
9 compile:
10    vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp
11    ALU.vp Top.v Execute.if.sv Execute.tb.sv Execute.test_top.sv
12
13 simulate:
14    vsim -l sim.log -c -novopt Execute_test_top -do "run -all"
15
16 clean:
17    rm -rf work mti_lib transcript modelsim.ini
```

可以看到，我们将包括删除文件在内的一共 4 条命令以某种方式写入到 Makefile 这个文件中去了。

#### ③ Makefile 的使用

保存 Makefile 文件之后，在当前目录下就多出了一个 Makefile 文件。在 Terminal 中键入

make

之后可以看到仿真开始执行直到结束。

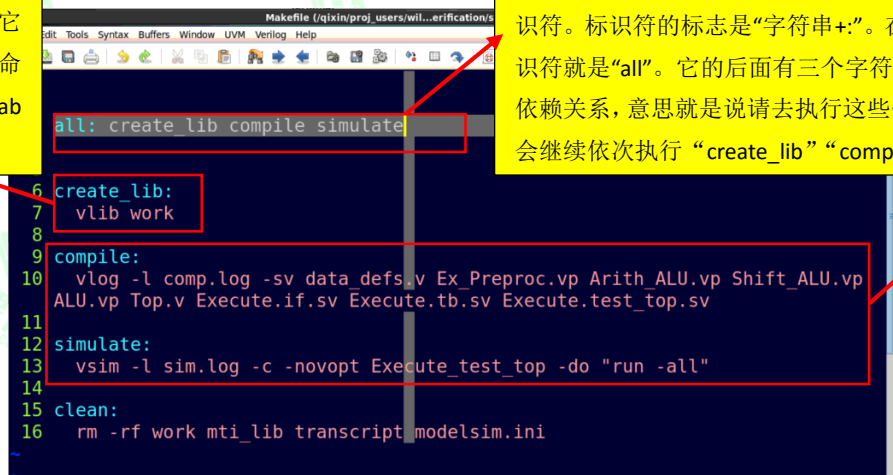
以后每次执行仿真，只要在该目录下执行“make”，即可自动执行仿真。

#### ④ Makefile 简介

以上面的 Makefile 文件为例，来看看这些代码是什么意思。

当执行“make”命令时，该命令会在当前目录下寻找名为“Makefile”的文件并执行其中的命令

先找到“create\_lib”，执行它下面一行的“vlib work”命令。注意第 7 行的开头是 tab 符，而不是其他空白符。



```
1
2
3 all: create lib compile simulate
4
5
6 create_lib:
7     vlib work
8
9 compile:
10    vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp
11    ALU.vp Top.v Execute.if.sv Execute.tb.sv Execute.test_top.sv
12
13 simulate:
14    vsim -l sim.log -c -novopt Execute_test_top -do "run -all"
15
16 clean:
17    rm -rf work mti_lib transcript modelsim.ini
```

如果 make 命令后没有跟参数，则会执行遇到的第一个标识符。标识符的标志是“字符串+:”。在本例中，第一个标识符就是“all”。它的后面有三个字符串，就是这个 all 的依赖关系，意思就是说请去执行这些依赖关系，也就是说会继续依次执行“create\_lib”“compile”“simulate”。

然后在分别找到“compile”“simulate”执行它们下面的命令

也就是说，如果执行的是“make”命令，则 Makefile 中的“all”“create\_lib”“compile”“simulate”下面的命令会被执行，最终达到自动执行的目的。

当然，如果想要执行这些命令中的某一条，则在 make 之后加上标识符就可以了，例如“make compile”，则只会执行

```
9 compile:
10 vlog -l comp.log -sv data_defs.v Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp
   ALU.vp Top.v Execute.if.sv Execute.tb.sv Execute.test_top.sv
```

也就是编译命令。其他情况类比，在本例中“make”和“make all”的效果是相同的。

同理，我们可以直接使用“make clean”来实现清理工作。

4. 认真阅读《E 课网 UVM 实战培训 SVV 实验 DUT 手册.pdf》文档，熟悉 DUT 的功能。弄清楚 DUT 的接口信号。可通过“Top.v”文件查看。
5. 认真阅读“Execute.if.sv”“Execute.tb.sv”以及“Execute.test\_top.sv”三个文件。弄清楚以下问题：
  - 什么是 Interface，Interface 的作用以及编写方法
  - 什么是 program，program 的作用以及编写方法
  - Testbench 和 DUT 在什么地方连接起来的，以及如何连接起来的

### 实验问题：

请将以下问题以回帖的方式在论坛上作答。

1. 语句

```
repeat (5) @(Execute.cb)
```

做了什么事情？（5 分）

答：

2. 在 reset task 中实现了什么功能以及有什么样的时序特性？（10 分）

答：

3. 在 Testbench 中包含可很多个 task，请根据在课堂上学习到的 SystemVerilog Testbench 的层次结构，指出这些 task 分别对应着 Testbench 的哪一层。（20 分）

答：

4. 尝试修改 Testbench 代码，使其完成下面的功能：

- a) Testbench 运行 5 次；并且（5 分）
- b) 每次向 DUT 发送 10 条操作指令。（5 分）

答：

5. 修改 Testbench，尝试向 DUT 输入操作指令，使 DUT 执行以下功能：

- a) 带符号的加法操作：在 immp\_reg\_n\_op = 1 时，实现 aluout = -45 + 60 操作（5 分）
- b) 算术左移操作：将操作数-23 向左移动 5 位，其中 5 是一个立即值。（5 分）
- c) 将数据 32'hbaad 写入存储器中（从 mem\_data\_write\_out 端口输出）（5 分）
- d) memory/src2 读操作（LOAD 操作）

- i. Load sign-extended byte[7:0] （10 分）

- ii. Load sign-extended half-word[15:0] (10 分)
- iii. Load entire 32 bits[31:0] (10 分)
- iv. Load half-word[15:0] to most significant word of output (10 分)

答: