

Synthesis: by Genus

EE291F

Speaker Yuxuan Li (liyx22025@shanghaitech.edu.cn)

VSPLab@SIST



上海科技大学

ShanghaiTech University

Contents

目录

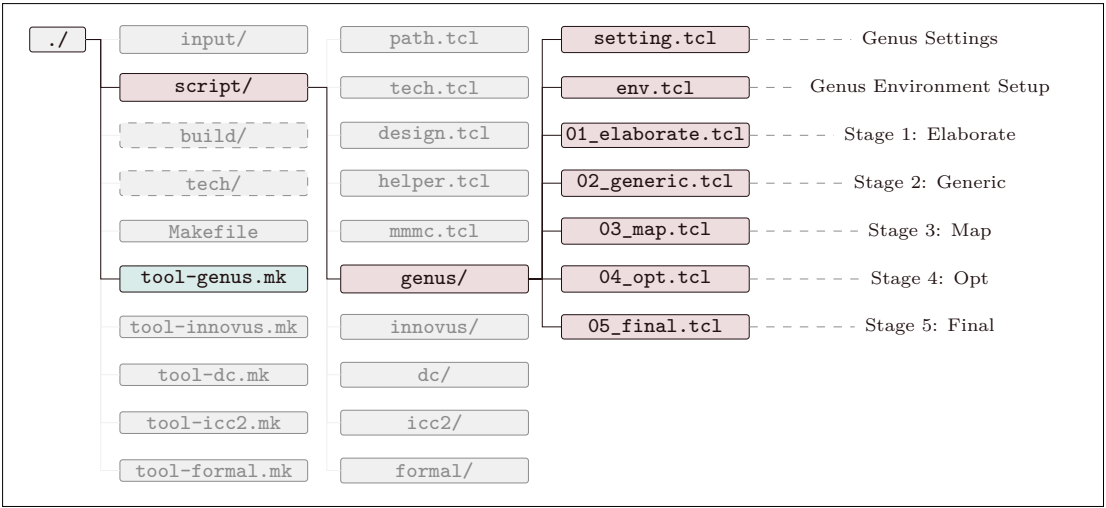
1 Genus：环境准备

2 MMMC

3 Genus：生成通用网表

4 Genus：生成工艺网表

Genus 文件组织总览



Genus：环境准备

Genus 的启动

Genus 可以通过以下方式从命令行启动

```
1 genus
```


立志成才 报国裕民

在构建目录中启动!

但这样做会产生两个问题

- Genus 运行后会在启动目录产生 `genus.log` 和 `genus.cmd`，污染源码目录。
- Genus 的日志文件名固定，重复运行会得到 `log1` `log2` 和 `cmd1` `cmd2`，不便于溯源。

数字后端的 EDA 软件普遍有在启动目录不受控地产生大量文件的倾向，这是需要约束的!

立志成才 报国裕民

在构建目录中启动！

总是在一个固定的构建目录中启动工具！这样所有的产物都落在构建目录里

```
cd build/genus
genus
```

这个流程我们可以用 Makefile 的方式固定下来！

立志成才 报国裕民

tool-genus.mk

创建一个独立的 tool-genus.mk，并在主 Makefile 中包含它

```
include tool-genus.mk
```

定义 Genus 的命令名称

```
GENUS:=genus
```

定义 Genus 的构建目录和日志文件，**所有日志文件都用时间戳命名**

```
GENUS_BUILD:=${BUILD_DIR}/genus
GENUS_CMD:=${GENUS_BUILD}/${RUN_STAMP}-genus.cmd
GENUS_LOG:=${GENUS_BUILD}/${RUN_STAMP}-genus.log
GENUS_COMMIT:=${GENUS_BUILD}/${RUN_STAMP}-git-commit.txt
```

tool-genus.mk

实现构建目录的创建和删除

```
1 ${GENUS_BUILD}: | ${BUILD_DIR}
2     mkdir -p $@
```

```
1 genus-clean:
2     rm -r -v -I ${GENUS_BUILD}
```

定义 Genus 的交互启动目标 genus-shell

```
genus-shell: | ${GENUS_BUILD}
    echo "${GIT_COMMIT}" >> $(abspath ${GENUS_COMMIT})
    echo "${RUN_COMMAND}" >> $(abspath ${GENUS_COMMIT})
    cd ${GENUS_BUILD} && ${GENUS} \
        -log "$(abspath ${GENUS_LOG}) $(abspath ${GENUS_CMD})"
```

- `${GENUS_COMMIT}` 文件用于记录本次运行时仓库的版本号和启动命令，便于溯源。
- `${GENUS_LOG}` 和 `${GENUS_CMD}` 即 `.log` 和 `.cmd` 文件，通过 `-log` 选项指定。
- `cd ${GENUS_BUILD} && ${GENUS}` 代表先进入构建目录再启动工具，`&&` 用于连接两条顺序执行的命令。**不可以拆成两行**，Makefile 对于每行命令会启用一个独立的 Shell 来执行。
- `$(abspath ...)` 是 Makefile 获取绝对路径的函数：切换目录后，相对路径不再正确了。

Genus 命令帮助

在 Genus 中，你可以通过三种方式获得关于一条命令的用法信息，以 syn_generic 为例。

查看完整手册

```
man syn_generic
```

查看简要帮助

```
help syn_generic
```

也可以使用

```
syn_generic -help
```

交互式输入 Tcl 命令是很低效的，你可以编写脚本！**那如何让工具在启动时运行某个脚本呢**？

定义 Genus 的脚本运行目标 `genus-run`

```
genus-run: | ${GENUS_BUILD}
    echo "${GIT_COMMIT}" >> $(abspath ${GENUS_COMMIT})
    echo "${RUN_COMMAND}" >> $(abspath ${GENUS_COMMIT})
    cd ${GENUS_BUILD} && ${GENUS} \
        -log "$(abspath ${GENUS_LOG}) $(abspath ${GENUS_CMD})" \
        -batch -files $(abspath ${GENUS_MAIN})
```

其中，`-batch` 指定运行完自动退出，`-files` 指定要运行的脚本

你可以通过 `make genus-run GENUS_MAIN=...` 的方式，从命令行指定具体要运行哪一个脚本！

每个阶段都是一个独立可运行的脚本！它们的执行都可以由 `make genus-run` 派生实现

```
genus-elaborate:
    $(MAKE) genus-run GENUS_MAIN=${GENUS_SCRIPT}/01_elaborate.tcl
genus-generic:
    $(MAKE) genus-run GENUS_MAIN=${GENUS_SCRIPT}/02_generic.tcl
genus-map:
    $(MAKE) genus-run GENUS_MAIN=${GENUS_SCRIPT}/03_map.tcl
genus-opt:
    $(MAKE) genus-run GENUS_MAIN=${GENUS_SCRIPT}/04_opt.tcl
genus-final:
    $(MAKE) genus-run GENUS_MAIN=${GENUS_SCRIPT}/05_final.tcl
```

完整流程就是顺次执行每个阶段

```
genus:
    $(MAKE) genus-elaborate
    $(MAKE) genus-generic
    $(MAKE) genus-map
    $(MAKE) genus-opt
    $(MAKE) genus-final
```

env.tcl

Genus 所有可以独立运行的脚本都需要统一加载一些配置，这可以写在 env.tcl 内

```
set TOOL genus
source [file join [file dirname [info script]] .. path.tcl]
source [file join ${SCRIPT_DIR} tech.tcl]
source [file join ${SCRIPT_DIR} design.tcl]
source [file join ${SCRIPT_DIR} helper.tcl]
source [file join ${GENUS_SCRIPT} setting.tcl]
set_multi_cpu_usage -local_cpu 8 -verbose
```

- TOOL 定义工具名，后续编写的某些脚本需要对不同工具做出不同行为。
- setting.tcl 存储 Genus 工具私有的配置。

请确保 path.tcl、tech.tcl、design.tcl、helper.tcl 以及 setting.tcl 已正确创建！

MMMC

MMMC 是什么

MMMC的全称是**多模式多角**（Multi-Mode Multi-Corner）

- **多模式**（Multi Mode）：对应 SDC 文件。设计应在**各工作场景的约束**下都能工作[1]。
- **多角**（Multi Corner）：对应 PVT 角和 RC 角。设计应在**各工艺和环境偏差**下都能工作。

课程案例中的 MMMC 相当简单：我们只会使用一个模式（正常模式）和两个角（快/慢）。

[1]例如，对于一些具有可测性设计的 RTL，可能会有“正常模式”和“测试模式”两份 SDC。

立志成才 报国裕民

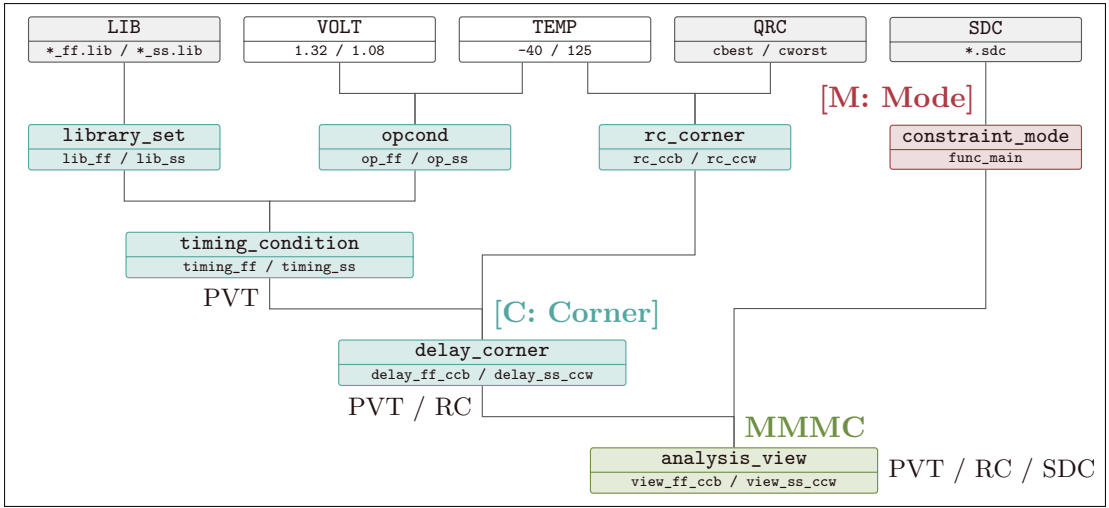
MMMC 是什么

MMMC 通常由一个单独的 Tcl 脚本维护，Genus 和 Innovus 可以共用一套 MMMC 脚本！

- MMMC 脚本放在 `script/mmmc.tcl`，作为 Genus 和 Innovus 可以共用的文件。
- MMMC 脚本内的概念很杂，但只要弄清各个概念之间的逻辑关系，就很清楚了！

立志成才 报国裕民

MMMC 的名称层级


LIB
*_ff.lib / *_ss.lib
VOLT
1.32 / 1.08
TEMP
-40 / 125
QRC
cbest / cworst
SDC
*.sdc
[M: Mode]
library_set
lib_ff / lib_ss
opcond
op_ff / op_ss
rc_corner
rc_ccb / rc_ccw
constraint_mode
func_main
timing_condition
timing_ff / timing_ss
PVT
[C: Corner]
delay_corner
delay_ff_ccb / delay_ss_ccw
PVT / RC
MMMC
analysis_view
view_ff_ccb / view_ss_ccw
PVT / RC / SDC

根据工具选择 SDC 文件

```
1 if {${TOOL} eq "genus"} {
2     set SDC_MAIN ${SDC_FILE}
3 }
```

Genus 加载的 SDC 文件直接来自设计的输入。但通常来说，综合流程结束时，工具一般会再输出一份 SDC 以反映过程中设计约束的变化。届时，Innovus 需要在此处加载不同的 SDC 文件。

mmmc.tcl

create_library_set 由 LIB 文件创建 library_set，名称记为 lib_*

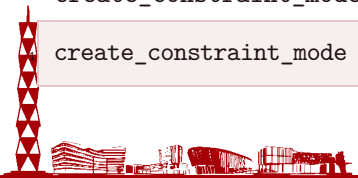
```
create_library_set -name lib_ff -timing ${LIB_LIST_FF}
create_library_set -name lib_ss -timing ${LIB_LIST_SS}
```

create_rc_corner 由 QRC 文件和温度创建 rc_corner，名称记为 rc_*

```
create_rc_corner -name rc_ccb -temperature ${TEMP_L} -qrc_tech ${QRC_CCB}
create_rc_corner -name rc_ccw -temperature ${TEMP_H} -qrc_tech ${QRC_CCW}
```

create_constraint_mode 由 SDC 文件创建 constraint_mode，名称记为 func_*

```
create_constraint_mode -name func_main -sdc_files [list ${SDC_MAIN}]
```



create_opcond 组合电压和温度创建 opcond，名称记为 op_*

```
create_opcond -name op_ff -process 1 -voltage ${VOLT_H} -temperature ${TEMP_L}
create_opcond -name op_ss -process 1 -voltage ${VOLT_L} -temperature ${TEMP_H}
```

create_timing_condition 组合 library_set 和 opcond，名称记为 timing_*

```
create_timing_condition -name timing_ff -library_sets lib_ff -opcond op_ff
create_timing_condition -name timing_ss -library_sets lib_ss -opcond op_ss
```

create_delay_corner 组合 timing_condition 和 rc_corner，名称记为 delay_*

```
create_delay_corner -name delay_ff_ccb \
    -timing_condition timing_ff -rc_corner rc_ccb
create_delay_corner -name delay_ss_ccw \
    -timing_condition timing_ss -rc_corner rc_ccw
```

create_analysis_view 组合 constraint_mode 和 delay_corner，名称记为 view_*

```
create_analysis_view -name view_ff_ccb \
    -constraint_mode func_main -delay_corner delay_ff_ccb
create_analysis_view -name view_ss_ccw \
    -constraint_mode func_main -delay_corner delay_ss_ccw
```

set_analysis_view 声明 setup 与 hold 各使用哪些 analysis_view

```
1 set VIEWS_TS {view_ss_ccw view_ff_ccb}
2 set VIEWS_TH {view_ss_ccw view_ff_ccb}
3 set_analysis_view -setup ${VIEWS_TS} -hold ${VIEWS_TH}
```

Genus：生成通用网表

01_elaborate.tcl

在每个阶段开始，要引入 env.tcl、创建阶段目录、定义阶段数据库

```
source [file join [file dirname [info script]] env.tcl]
set STAGE_NAME  01_elaborate
set STAGE_DIR   [stage_make_dir ${GENUS_BUILD} ${STAGE_NAME}]
set STAGE_DB    [file join ${STAGE_DIR} ${DESIGN}.db]
```

- 阶段脚本可以独立运行，因此每个阶段都需要独立引入 env.tcl 加载环境配置。
- ${STAGE_DIR} 定义阶段的构建目录，例如 build/genus/01_elaborate。
- ${STAGE_DB} 定义阶段的数据库，下一阶段会读取该数据库以恢复状态。

`stage_make_dir` 在公共的 `helper.tcl` 中定义，用于创建阶段目录

```
proc stage_make_dir {build_dir stage_name} {
    set stage_dir [file join ${build_dir} ${stage_name}]
    file delete -force ${stage_dir}
    file mkdir ${stage_dir}
    return ${stage_dir}
}
```

该函数的行为是，先删除（`file delete -force`），后创建（`file mkdir`），确保干净。

01_elaborate.tcl

设置日志详细程度，set_db 用于设置 Genus 数据库属性

```
set_db information_level 9
```

设置时间单位为 ns（默认为 ps），使其与 SDC 和 Innovus 保持一致

```
set_db ui_units_timing_reporting      1ns
```

设置电容单位为 pf（默认为 ff），使其与 SDC 和 Innovus 保持一致

```
set_db ui_units_capacitance_reporting 1pf
```

设置后续三个阶段 syn_generic、syn_map、syn_opt 的计算努力

```
set_db syn_generic_effort ${SYN_GENERIC_EFFORT}
set_db syn_map_effort     ${SYN_MAP_EFFORT}
set_db syn_opt_effort     ${SYN_OPT_EFFORT}
```

这些变量定义在 setting.tcl 中

```
set SYN_GENERIC_EFFORT high
set SYN_MAP_EFFORT high
set SYN_OPT_EFFORT high
```

01_elaborate.tcl

关闭一些和 DFT 相关的设置，该设计没有做可测性设计

```
set_db use_scan_seqs_for_non_dft false
```

关闭一些优化，这些优化可能会妨碍 Formality 形式化验证

```
set_db optimize_constant_feedback_seqs false
set_db optimize_constant_0_flops false
set_db optimize_merge_flops false
```

Genus 和 DC 都会将优化过程中的变换记录下来，但相应文件格式只能被各自的形式化验证工具读取。由于该课程只提供 Formality，关闭这些优化可以避免其网表无法通过形式化验证[2]。

[2]当然，这门课也只要求对 DC 的网表做形式化验证，因此你可以不禁用这些优化。

立志成才 报国裕民

01_elaborate.tcl

导入 MMMC

```
read_mmmc [file join ${SCRIPT_DIR} mmmc.tcl]
```

导入 LEF[3]

```
read_physical -lefs ${LEF_LIST}
```

导入 RTL

```
read_hdl ${RTL_LIST}
```

[3]逻辑综合并不需要 LEF，但 MMMC 中读入的 QRC 需要依赖此处的 LEF，否则会报错。

01_elaborate.tcl

elaborate 用于将 RTL 转换为通用网表，**这是第一阶段的核心命令**

```
elaborate ${DESIGN}
```

init_design 用于初始化项目

```
init_design
```

保护时钟网络

```
1 set_dont_touch_network [get_ports ${PORT_CLK}]
```

保护复位网络

```
1 set_dont_touch_network [get_ports ${PORT_RST}]
```

综合阶段不应触碰时钟网络和复位网络，它们需要在 CTS 阶段专门处理。

禁用一批不适合用于综合的单元

```
1 set DONT_USE_CELLS [c_get_base_cell_names ${DONT_USE_CELL_PATTERNS}]
2 set_db [get_db base_cells ${DONT_USE_CELLS}] .dont_use true
```

- ${DONT_USE_CELL_PATTERNS} 是模式列表，包含 * 和 ? 的通配符，在 setting.tcl 定义。
- ${DONT_USE_CELLS} 是名称列表，它是通过在单元库中进行模式匹配查询得到的。
- c_get_base_cell_names 是一个定义在 helper.tcl 中的辅助函数。

01_elaborate.tcl

${DONT_USE_CELL_PATTERNS} 在 setting.tcl 中定义

```
set DONT_USE_CELL_PATTERNS [list \
    {*DEL*} \
    {*ANT*} \
    {*FILL*} \
    {*TAP*} \
    {*CLK*} \
    {*GCK*} \
    {*TIE*} \
    {*X20A?9} \
    {*X05A?9} \
]
```

禁用单元

表: 禁用单元

模式	说明
DEL	延时单元（delay）
ANT	缓解导线天线效应的单元（antenna）
FILL	填充行内空白的单元（fill）
TAP	提供阱连接的单元（tap）
CLK	时钟树单元（clock tree）
GCK	时钟门控单元（clock gating）
TIE	逻辑常数 0/1 单元（tie）
*X20A?9	最大驱动强度单元
*X05A?9	最小驱动强度单元

c_get_base_cell_names 在 helper.tcl 中定义，输入模式列表，输出名称列表

```tcl
proc c_get_base_cell_names {patterns} {
    set names [list]
    foreach pattern ${patterns} {
        foreach cell [get_db base_cells ${pattern}] {
            lappend names [get_db ${cell} .name]
        }
    }
    return [lsort -unique ${names}]
}
```

检查是否有未解析的引用

```
check_design -unresolved > [file join ${STAGE_DIR} check_design.rpt]
```

导出网表

```
write_netlist > [file join ${STAGE_DIR} netlist.v]
```

保存数据库

```
write_db -to_file ${STAGE_DB}
```

Genus：生成工艺网表

02_generic.tcl

第二阶段的文件头

```
source [file join [file dirname [info script]] env.tcl]
set STAGE_NAME  02_generic
set STAGE_DIR   [stage_make_dir ${GENUS_BUILD} ${STAGE_NAME}]
set STAGE_DB    [file join ${STAGE_DIR} ${DESIGN}.db]
```

读取上一阶段的数据库

```
read_db [file join ${GENUS_BUILD} 01_elaborate ${DESIGN}.db]
```

syn_generic 用于优化通用网表，**这是第二阶段的核心命令**

```
syn_generic
```

02_generic.tcl

生成报告包

```
write_snapshot -directory ${STAGE_DIR} -tag syn_generic
report_summary -directory ${STAGE_DIR}
```

导出网表

```
write_netlist > [file join ${STAGE_DIR} netlist.v]
```

保存数据库

```
write_db -to_file ${STAGE_DB}
```

03_map.tcl

第三阶段的文件头

```
source [file join [file dirname [info script]] env.tcl]
set STAGE_NAME  03_map
set STAGE_DIR   [stage_make_dir ${GENUS_BUILD} ${STAGE_NAME}]
set STAGE_DB    [file join ${STAGE_DIR} ${DESIGN}.db]
```

读取上一阶段的数据库

```
read_db [file join ${GENUS_BUILD} 02_generic ${DESIGN}.db]
```

syn_map 用于生成并优化工艺网表，**这是第三阶段的核心命令**

```
syn_map
```

03_map.tcl

生成报告包

```
write_snapshot -directory ${STAGE_DIR} -tag syn_map
report_summary -directory ${STAGE_DIR}
```

导出网表

```
write_netlist > [file join ${STAGE_DIR} netlist.v]
```

保存数据库

```
write_db -to_file ${STAGE_DB}
```

04_opt.tcl

第四阶段的文件头

```
source [file join [file dirname [info script]] env.tcl]
set STAGE_NAME  04_opt
set STAGE_DIR   [stage_make_dir ${GENUS_BUILD} ${STAGE_NAME}]
set STAGE_DB    [file join ${STAGE_DIR} ${DESIGN}.db]
```

读取上一阶段的数据库

```
read_db [file join ${GENUS_BUILD} 03_map ${DESIGN}.db]
```

syn_opt 用于优化工艺网表，**这是第四阶段的核心命令**

```
syn_opt
```

04_opt.tcl

生成报告包

```
write_snapshot -directory ${STAGE_DIR} -tag syn_opt
report_summary -directory ${STAGE_DIR}
```

导出网表

```
write_netlist > [file join ${STAGE_DIR} netlist.v]
```

保存数据库

```
write_db -to_file ${STAGE_DB}
```

05_final.tcl

第五阶段的文件头

```
source [file join [file dirname [info script]] env.tcl]
set STAGE_NAME  05_final
set STAGE_DIR   [stage_make_dir ${GENUS_BUILD} ${STAGE_NAME}]
set STAGE_DB    [file join ${STAGE_DIR} ${DESIGN}.db]
```

读取上一阶段的数据库

```
read_db [file join ${GENUS_BUILD} 04_opt ${DESIGN}.db]
```

许多 05_final 的产物实际上和上一阶段完全相同，它更多是作为产生正式交付的架构定位。

05_final.tcl

生成报告包[4]

```
write_snapshot -innovus -directory ${STAGE_DIR} -tag genus
report_summary -directory ${STAGE_DIR}
```

导出网表

```
write_netlist > ${GENUS_NETLIST}
```

导出约束

```
write_sdc -constraint_mode func_main \
    -exclude {set_dont_touch_network set_dont_use} > ${GENUS_SDC_MAIN}
```

[4]选项 -innovus 代表快照使用 Innovus 兼容格式，但实际上这里的产物并未被 Innovus 使用。

05_final.tcl

通常来说，我们会在综合流程结束后导出一份 SDC，尽管其内容和输入的 SDC 没有本质差别

- set_dont_touch_network 的约束应移除，否则后续 CTS 无法做优化。
- set_dont_use 的约束应移除，实现阶段可以根据实际需要重新设置禁用单元。

${GENUS_NETLIST} 和 ${GENUS_SDC_MAIN} 是 Genus 的交付产物，在 design.tcl 中定义

```
set GENUS_FINAL_DIR [file join ${GENUS_BUILD} 05_final]
set GENUS_NETLIST   [file join ${GENUS_FINAL_DIR} ${DESIGN}_syn.v]
set GENUS_SDC_MAIN  [file join ${GENUS_FINAL_DIR} ${DESIGN}_syn_func_main.sdc]
```

保存数据库

```
write_db -to_file ${STAGE_DB}
```


立志成才 报国裕民

Thank You

谢谢观看