Lab 开源 EDA 工具全流程使用复现

目标

通过开源 RTL 综合器 yosys 对 RTL 设计进行逻辑综合，使用开源静态时序分析工具 iSTA 评估 RTL 设计中的关键时序路径

基于生成的网表，使用 iEDA 工具，物理后端设计流程并分析结果。有关 iEDA 的使用流程完全参照 iEDA 的 user_guide，实验中遇到的问题几乎都可以从中找到答案。

环境要求

-Yosys

-iEDA（需要 Ubuntu 20.04.5 LTS 及以上 Ubuntu 系统）

-Smic180 PDK

实验步骤

linux 环境设置

键入命令：`export LD_LIBRARY_PATH=$(echo $LD_LIBRARY_PATH | tr ':' '\n' | grep -v 'xilinx' | paste -sd ':')`

然后键入以下命令：`echo $LD_LIBRARY_PATH`

此时$LD_LIBRARY_PATH 中不应包含带有“Xilinx”关键词的路径

逻辑综合实验目录/mnt/home/students/zhaokang/Lab/yosys-sta

（这是提供参考的文件，请不要直接修改，需要先拷贝到自己的目录下）

.
├── cpu //这个 cpu 的 RTL 设计文件夹
├── bin //iSTA 可执行文件
| ├── iSTA
| └── libyaml-cpp.so.0.6
├── Makefile //整个流程的 makefile 脚本
├── smic180 //smic180 工艺库（部分）
├── README.md
├── result //跑完综合后会自动生成该文件夹，所有综合后的 log、rpt 都会放在这里
| └── cpu-10MHz //自动命名格式是 DESIGNNAME-xMHz

├── sta.tcl //make sta 命令调用的 sta 脚本
└── yosys.tcl //make syn 命令调用的 syn 脚本

使用 yosys 工具完成逻辑综合

1）在 CPU 顶层外围包 IO/PAD

提供的 CPU 代码的最顶层是 cpu.v，不包含 IO，真实芯片中，需要通过 IO cell 和芯片外部进行互连。IO 的作用一是完成 core 电平和 IO 电平转换，例如 core 是 1.8V，IO 是 3.3V；二是完成特殊逻辑功能，例如双向 IO、三态 IO；三是提供 ESD 保护。有关基于 smic180nm 工艺的 IO/PAD 的具体信息需要参考手册：smic180 路径下的 SMIC_SP018_IO_DataBook_Ver1p3.pdf 文档。

在 CPU 顶层外围包 IO/PAD 的具体方法是建立新的顶层文件，比如 full_chip.v，在其中例化相应的 IO cell 并例化原先顶层模块，把原先顶层模块的 input/output 通过 IO cell 拉到芯片外部。在 cpu 文件夹下编写 full_chip.v 文件如下：

```
module full_chip (
    input rst_,
    input clock,
    input [7:0] data_out,

    output  rd,
    output  wr,
    output [7:0] data_in,
    output halt,
    output [4:0] addr
);
    wire clock_pin;
    wire rst_pin;
    wire [7:0] data_out_pin;

    wire [7:0] data_in_pin;
    wire [4:0] addr_pin;
    wire rd_pin;
    wire wr_pin;

    PI  u_pad_clock ( .PAD(clock),  .C(clock_pin));
    PI      u_pad_rst_ ( .PAD(rst_),      .C(rst_pin));

    PI  u_pad_data_out7 ( .PAD(data_out[7]),    .C(data_out_pin[7]));
    PI  u_pad_data_out6 ( .PAD(data_out[6]),    .C(data_out_pin[6]));
    PI  u_pad_data_out5 ( .PAD(data_out[5]),    .C(data_out_pin[5]));
    PI  u_pad_data_out4 ( .PAD(data_out[4]),    .C(data_out_pin[4]));
    PI  u_pad_data_out3 ( .PAD(data_out[3]),    .C(data_out_pin[3]));
    PI  u_pad_data_out2 ( .PAD(data_out[2]),    .C(data_out_pin[2]));
    PI  u_pad_data_out1 ( .PAD(data_out[1]),    .C(data_out_pin[1]));
    PI  u_pad_data_out0 ( .PAD(data_out[0]),    .C(data_out_pin[0]));
```

```
    PO8  u_pad_data_in7 ( .I(data_in_pin[7]),   .PAD(data_in[7]));
    PO8  u_pad_data_in6 ( .I(data_in_pin[6]),   .PAD(data_in[6]));
    PO8  u_pad_data_in5 ( .I(data_in_pin[5]),   .PAD(data_in[5]));
    PO8  u_pad_data_in4 ( .I(data_in_pin[4]),   .PAD(data_in[4]));
    PO8  u_pad_data_in3 ( .I(data_in_pin[3]),   .PAD(data_in[3]));
    PO8  u_pad_data_in2 ( .I(data_in_pin[2]),   .PAD(data_in[2]));
    PO8  u_pad_data_in1 ( .I(data_in_pin[1]),   .PAD(data_in[1]));
    PO8  u_pad_data_in0 ( .I(data_in_pin[0]),   .PAD(data_in[0]));

    PO8 u_pad_addr4 ( .I(addr_pin[4]),    .PAD(addr[4]));
    PO8 u_pad_addr3 ( .I(addr_pin[3]),    .PAD(addr[3]));
    PO8 u_pad_addr2 ( .I(addr_pin[2]),    .PAD(addr[2]));
    PO8 u_pad_addr1 ( .I(addr_pin[1]),    .PAD(addr[1]));
    PO8 u_pad_addr0 ( .I(addr_pin[0]),    .PAD(addr[0]));

    PO8 u_pad_rd ( .I(rd_pin),      .PAD(rd));
    PO8 u_pad_wr ( .I(wr_pin),      .PAD(wr));

    cpu u_cpu(
        .rst_       (rst_pin        ),
        .clock      (clock_pin      ),
        .data_out   (data_out_pin   ),
        .data_in    (data_in_pin    ),
        .addr       (addr_pin       ),
        .rd         (rd_pin         ),
        .halt       (halt           ),
        .wr         (wr_pin         )
    );
endmodule
```

2）修改 Makefile 文件

本实验使用 Makefile 脚本调用 yosys 工具做逻辑综合，需要根据情况对 Makefile 脚本做修改。

```
PROJ_PATH = $(shell pwd)

DESIGN ?= full_chip
SDC_FILE ?= $(PROJ_PATH)/cpu/CPU.sdc
#RTL_FILES ?= $(shell find $(PROJ_PATH)/example -name "*.v")
RTL_FILES ?= $(PROJ_PATH)/cpu/full_chip.v $(PROJ_PATH)/cpu/cpu.v $(PROJ
_PATH)/cpu/register.v $(PROJ_PATH)/cpu/mux.v $(PROJ_PATH)/cpu/counter.v
 $(PROJ_PATH)/cpu/control.v $(PROJ_PATH)/cpu/scale_mux.v $(PROJ_PATH)/c
pu/dffr.v $(PROJ_PATH)/cpu/alu.v
export CLK_FREQ_MHZ ?= 500
```

PROJ_PATH = $(shell pwd)一般无需修改，也就是将当前目录设置为 PROJ_PATH 的值，以便在脚本的之后部分使用。

DESIGN ?= full_chip，这里需要将该变量修改为 RTL 设计中顶层设计的名称。

SDC_FILE ?= $(PROJ_PATH)/cpu/CPU.sdc，这里需要将该变量修改为 sdc 文件路径。（不过综合时不会调用这个文件，该 sdc 文件是 sta 时用到的）

还需要将 RTL_FILES 变量的值修改为所有希望进行综合的.v 文件。在脚本的第 5 行还提供了另外一种给 RTL_FILES 变量赋值的办法，不过暂时使用#将其注释。这种办法是使用了 find 这个命令，find $(PROJ_PATH)/RTL -name "*.v"可以遍历该目录下所有子文件夹，找到所有后缀为.v 的文件。一般会把所有待综合的.v 文件都集中放在一个文件夹下，并使用这种方式给 RTL_FILES 变量赋值。

这里没有采用这种方法的原因是：很多人会把各模块 testbench 的.v 文件也放在和可综合模块相同的目录下，因此为了避免习惯上的冲突，干脆采用后面这种“枚举”所有.v 文件的方式给 RTL_FILES 变量赋值。

这里需要特别注意：在 RTL_FILES 的文件中必须包含一个名为 DESIGN 的 module

export CLK_FREQ_MHZ ?= 500，在这里你需要设定目标综合频率，注意单位是 MHz。

3）在当前目录下运行以下命令完成综合：

```
make syn
```

在这里可能会遇到各种各样报错，应当根据报错对设计进行修改，相关报错应当优先参考 yosys 的 github 网址下的 issues 部分。

4）阅读综合后的 log 和 rpt

在 result 目录下，可以看到综合后的 log 和 rpt 文件，以及最关键的 netlist 网表文件，该文件夹目录说明如下：

```
├── result
|   └── full_chip-500MHz
|       ├── full_chip.netlist.v - Yosys 综合的网表文件
|       ├── synth_check.txt  - Yosys 综合的检查报告，用户需仔细阅读并决定是否需要排除相应警告
|       ├── synth_stat.txt  - Yosys 综合的面积报告
|       └── yosys.log  - Yosys 综合的完整日志
```

大体来讲，synth_check.txt 文件应当是首先阅读的，应当确保里面没有“error”提示，此外文件中的“warning”也需要一一阅读。

Yosys 综合生成的网表文件是逻辑综合后得到的关键输出文件，网表是由标准单元组成的“门级”电路。有关标准单元的信息可以查看标准单元手册：smic180 目录下的 scc018ug_uhd_rvt.pdf 文档，有助于理解 full_chip.netlist.v 文件。

使用 iSTA 工具做简单的时序评估

1）修改 sdc 文件

```
set clk_port_name clock
set CLK_FREQ_MHZ 500
if {[info exists env(CLK_FREQ_MHZ)]} {
  set CLK_FREQ_MHZ $::env(CLK_FREQ_MHZ)
} else {
  puts "Warning: Environment CLK_FREQ_MHZ is not defined. Use $CLK_FREQ
_MHZ MHz by default."
}
set clk_io_pct 0.2

set clk_port [get_ports $clk_port_name]
create_clock -name core_clock -period [expr 1000 / $CLK_FREQ_MHZ] $clk_
port
```

sdc 文件只需要修改第一行内容，这里需要特别注意：该时钟端口名称需要与设计文件保持一致

比如：顶层设计中如果时钟端口是 `input clock`，则 sdc 文件第一行应为 `set clk_port_name clock`

你可能注意到第 2 行似乎还设置了一个时序检查的目标频率，该参数还会在接下来被使用。不过，这里 3-6 行的内容是说：首先检查环境变量 `CLK_FREQ_MHZ` 是否存在：

- 如果存在，脚本会更新 `CLK_FREQ_MHZ` 为环境变量的值。这是通过使用 `env(CLK_FREQ_MHZ)`读取环境变量并将其值分配给变量 `CLK_FREQ_MHZ` 来实现的。
- 如果环境变量不存在，它会打印一条警告消息，指出环境变量 `CLK_FREQ_MHZ` 未定义，并将默认使用 500 MHz 的值。

因此可以看到，第 2 行只是设置了一个默认值，环境变量中的值将在 Makefile 文件中设置。

2）在当前目录下运行以下命令完成静态时序分析：

```
make sta
```

这里需要注意的是，由于 Makefile 中设定的文件依赖关系，如果不先做 `make syn` 直接 `make sta`，那么脚本也会自动先执行 `make syn` 命令完成综合。因此可以把 `make sta` 看作一键式的，完成逻辑综合+静态时序分析的命令。

此外 Makefile 脚本提供了清除 result 文件夹的命令，在目录下执行 `make clean` 就可以删除 result 文件夹。

3）阅读时序分析后的 log 和 rpt

├── result
│ └── full_chip-500MHz
│ ├── full_chip.cap - iSTA 的电容违例报告
│ ├── full_chip.fanout - iSTA 的扇出违例报告
│ ├── full_chip_hold.skew - iSTA 的 hold 模式下时钟偏斜报告
│ ├── full_chip.netlist.v - Yosys 综合的网表文件
│ ├── full_chip.rpt - iSTA 的时序分析报告, 包含 WNS, TNS 和时序路径
│ ├── full_chip_setup.skew - iSTA 的 setup 模式下时钟偏斜报告
│ ├── full_chip.trans - iSTA 的转换时间违例报告
│ ├── synth_check.txt - Yosys 综合的检查报告, 用户需仔细阅读并决定是否需要排除相应警告
│ ├── synth_stat.txt - Yosys 综合的面积报告
│ └── yosys.log - Yosys 综合的完整日志

由于本设计的约束脚本（.sdc 文件）没有设置 skew，max cap，max fanout 等信息，因此主要关注 full_chip.rpt 中的信息内容，其中 setup 违例应当在逻辑综合阶段重点关注。如果 slack 出现了负值，说明我们的设计“跑不到”设定的目标频率，要么优化设计，要么降低目标频率，重新综合。

i NOTICE 阅读 full_chip.rpt，查看 setup 违例相关信息，指出 slack 值是否为负？如果存在负值，应当降低目标频率重新跑综合。 提示：降低目标频率的办法是修改 makefile 文件第八行的内容：export CLK_FREQ_MHZ ?= 500，尝试把目标频率改为 200M/300M/400MHz，再对比阅读 full_chip.rpt 中 setup 违例的相关信息，slack 为负值说明我们的设计依然“跑不到”设定的那么高的目标频率。

物理后端实验目录/mnt/home/software/iEDA/iEDA

本次实验主要需要关注的是/mnt/home/software/iEDA/iEDA 目录下的 scripts 文件夹：

.
├── design //该文件夹下存放不同的设计
│ ├── ispd18
│ ├── smic180_full_chip //存放有本次 CPU 设计和 iEDA 运行脚本
│ └── sky130_gcd //iEDA 项目提供的例程，需要在它的基础上做修改
├── docker
│ ├── build_succeeded.tcl
│ ├── Dockerfile.base
│ ├── Dockerfile.demogcd
│ ├── Dockerfile.demohello
│ ├── Dockerfile.release

| └── update_img.sh
├── foundry //使用的工艺库
| ├── README.md
| ├── smic180 //本次设计使用的工艺库
| └── sky130
└── hello.tcl

以 design 目录下 smic180_full_chip 设计为例，目录体系如下：

.
├── iEDA //iEDA 可执行文件
├── iEDA_config
├── README.md
├── result //执行后端流程后，报告的存放位置，其中的 verilog 文件夹需要存放 netlist
├── run_iEDA_gui.py
├── run_iEDA.py //物理后端全流程的 python 运行脚本
├── run_iEDA.sh
└── script //物理后端各子步骤的运行脚本

使用脚本完成物理后端全流程

阅读 iEDA 的 user_guide 中“物理后端设计全流程运行”这一章节内容，追踪从 run_iEDA.py 脚本开始，调用的各个子步骤的脚本顺序，并从中找出需要修改的文件和内容，为自己所用，下面我会将关键步骤做简单提炼。

1）将本次 cpu 的 netlist 和 sdc 文件拷贝到相应位置

拷贝 sdc 文件到 iEDA/scripts/design/smic180_full_chip/result/verilog/

```
# 拷贝 sdc 文件到目录 ./scripts/design/smic180_full_chip/result/verilog/
cp <sdc_path>/*.sdc scripts/design/smic180_full_chip/result/verilog/.
```

需要将上述命令中的<sdc_path>做替换，换为实际的 sdc 文件目录。下面拷贝 Netlist 文件到目录 iEDA/scripts/design/smic180_full_chip/result/verilog/

```
# 拷贝 .v 文件到目录 ./scripts/design/smic180_full_chip/result/verilog/
cp <netlist_path>/full_chip.netlist.v ./scripts/design/smic180_full_chi
p/result/verilog/.
```

需要将上述命令中的<netlist_path>做替换，换为实际的网表文件目录。

2）修改 db_path_setting.tcl 文件

```
#============================================================
##   set lef path
#============================================================
set TECH_LEF_PATH "./../../foundry/smic180/scc018u_6m_1tm1.lef"
```

```
set LEF_PATH "./../../foundry/smic180/scc018ug_uhd_rvt.lef \
              ./../../foundry/smic180/SP018_V1p5_6MT.lef"

#set DEF_PATH "./result/netlist_result.def"
set VERILOG_PATH "./../../design/smic180_full_chip/result/verilog/full_
chip.netlist.v"

set LIB_PATH "./../../foundry/smic180/scc018ug_uhd_rvt_ss_v1p62_125c_ba
sic.lib \
              ./../../foundry/smic180/SP018_V1p4_max.lib"

set LIB_PATH_FIXFANOUT ${LIB_PATH}
set LIB_PATH_DRV ${LIB_PATH}
set LIB_PATH_HOLD ${LIB_PATH}
set LIB_PATH_SETUP ${LIB_PATH}

set SDC_PATH_BEFORE_CTS "./../../design/smic180_full_chip/result/verilo
g/full_chip.sdc"
set SDC_PATH "./../../design/smic180_full_chip/result/verilog/full_chi
p.sdc"
set SPEF_PATH ""
```

3）其他修改

完成上述修改后，用./run_iEDA.py 命令应当可以完成物理后端全流程。不过后端流程中，各子步骤参数的设置以及报告的阅读是十分关键的。

比如第一步 floorplan，run_iFP.tcl 文件中：

```
set DIE_AREA "0.0    0.0   1560   1600"
set CORE_AREA "200.48 200.48 1359.52 1399.52"
set PLACE_SITE uhd_CoreSite
set IO_SITE IOSite
set CORNER_SITE CornerSite

init_floorplan \
   -die_area $DIE_AREA \
   -core_area $CORE_AREA \
   -core_site $PLACE_SITE \
   -io_site $IO_SITE \
   -corner_site $CORNER_SITE
```

这里是设置 DIE_AREA 和 CORE_AREA，根据 iEDA 的 user_guide 中相关内容，DIE_AREA 指芯片版图 DIE 面积，单位是平方微米，CORE_AREA 指芯片版图 CORE 面积， 单位是平方微米，CORE 的面积为所有标准单元 ROW 的面积之和。这里设置参数时，应当需要参考前端设计结束后输出的 synth_stat.txt 文件，里面包含芯片的面积等参数。

诸如此类的修改还有许多，应当结合具体情况做修改。完成物理后端流程后，各子步骤报告和最后的版图在 ./result/report 目录下。

4）物理后端设计分步骤/全流程运行

参考 iEDA 的 user_guide 中物理后端设计全流程运行部分

注意：run_iEDA.py 脚本中使用的 iEDA 可执行文件是在当前 smic180_full_chip 目录下的 iEDA 文件，因此如果在/mnt/home/software/iEDA/iEDA 目录下重新编译构建了 iEDA 可执行文件，需要使用 `cp ./bin/iEDA scripts/design/smic180_full_chip/.`命令把新编译构建好的 iEDA 可执行文件拷贝过来

5）查看各阶段生成的报告和生成的版图（gds2 文件）

阅读 iEDA 的 user_guide 中对于各个步骤后生成报告的解读，理解生成的报告

对于生成的.gds2 文件，可以使用相关软件查看。除了 iEDA 自带的 gui 界面，也可以使用 Klayout 等软件打开 gds2 文件查看，效果如下：