中国科学院大学高级计算机系统结构
课程 Project 研究报告

周盈坤 201918013229046
zhouyingkun15@mails.ucas.ac.cn

1 开源 EDA 工具链 OpenRoad 的安装

首先吐槽一下OpenRoad-flow这个 EDA 工具链还是非常难安装的，我折腾了很久。因为我的系统是 Ubuntu，而 OpenRoad 是在 CentOS 操作系统的，虽说都是 Linux 系统，但是难免会有一些系统上的差异，而为了规避这种系统差异所带来的不必要差错，OpenRoad 的开发者采用了 docker 容器的方法。一开始我也是认为这样做比较妥当。首先使用 docker 和本机没有太大的区别，我之前好些实验就都是在 docker 上完成的。而且像我使用的时候都会设置共享目录，这样就不需要在 docker 容器和本地文件系统之间来回折腾了。其次能用 docker 我也会尽量用，因为这样对本地系统的环境影响最少。但是万万没有想到，我一开始就栽在了这个 docker 中 OpenRoad 的安装上。docker 编译前会自动下载所需要的各个软件，然后进行编译安装。主要有如下几个 package：

```
cmake gcc[/ clang] bison flex swig boost tcl 8.5 zlib
```

当然还有很多杂七杂八的软件包。其中形象最深的就是 `swig-4.0.0.tar.gz` 软件包。运行 docker 编译 OpenRoad 工具链的时候，就会卡在这里，Terminal 中的 log 显示如下：

```
Step 5/19 : RUN wget http://prdownloads.sourceforge.net/swig/swig-4.0.0.tar.gz &&
    tar -xf swig-4.0.0.tar.gz &&     cd swig-4.0.0 &&     ./configure &&     make -j$(
    nproc) &&     make install
 ---> Running in d4bb8554312c
--2020-05-22 05:40:30--  http://prdownloads.sourceforge.net/swig/swig-4.0.0.tar.gz
Resolving prdownloads.sourceforge.net (prdownloads.sourceforge.net)... 216.105.38.13
Connecting to prdownloads.sourceforge.net (prdownloads.sourceforge.net)
    |216.105.38.13|:80... connected.
HTTP request sent, awaiting response... Read error (Connection reset by peer) in
    headers.
Retrying.

--2020-05-22 05:40:36--  (try: 2)  http://prdownloads.sourceforge.net/swig/swig-4.0.0.
    tar.gz
Connecting to prdownloads.sourceforge.net (prdownloads.sourceforge.net)
    |216.105.38.13|:80... connected.
```

```
HTTP request sent, awaiting response... Read error (Connection reset by peer) in
    headers.
Retrying.

# repeat trying many times ...

--2020-05-22 05:42:37--  (try:17)  http://prdownloads.sourceforge.net/swig/swig-4.0.0.
    tar.gz
Connecting to prdownloads.sourceforge.net (prdownloads.sourceforge.net)
    |216.105.38.13|:80... connected.
HTTP request sent, awaiting response... Read error (Connection reset by peer) in
    headers.
Retrying.
```

这是我写报告的时候再次执行的，目的就是重现 docker 执行的效果。之前 docker 执行到这里因为 `swig-4.0.0.tar.gz` 软件包始终下载不了，就会退出，并报错`wget Unable to establish SSL connection`。这跟家里的网络有关，每个人的情况都会不一样，有的人会非常顺利，有的人比如我就非常痛苦。可能我家的网络确实有问题吧，下载三四十兆的包都需要一两个小时，等得我焦头烂额。顺便说一句，OpenRoad 工具链源代码从 Github 上下载也特别费劲，总共 1G 多的文件，我特意挑了网速非常好的早上，也下了好久（貌似 Github 在中国地区目前就是这么一个情况，也有一些同学在群里反应）。

如果迈过了 `swig-4.0.0.tar.gz` 这道关卡（网络好的时候）。比如上面 log 中提示的我为了写这个报告而重新试着用 docker 下载编译安装 OpenRoad 工具链的时间–`2020-05-22 05:42:37`. 终于在第 18 次 try 之后成功连接并缓慢下载好了。docker 继续往下执行，知道下述 Terminal 中 log 显示的 `Step 3/14` 出现了 Error（如下 log 第 22 行）。并且在第 23 行报出`returned a non-zero code: 1`，出现异常退出了，但此时显然没有执行完，因为才执行到 14 步中的第 3 步。一个莫名其妙的错误?!

```
Sending build context to Docker daemon  859.4MB
Step 1/14 : FROM centos:centos7 AS base-dependencies
 ---> b5b4d78bc90c
Step 2/14 : LABEL maintainer="Abdelrahman Hosny <abdelrahman_hosny@brown.edu>"
 ---> Running in 609e56aba820
Removing intermediate container 609e56aba820
 ---> 8d8303883606
Step 3/14 : RUN yum group install -y "Development Tools"     && yum install -y https
    ://centos7.iuscommunity.org/ius-release.rpm     && yum install -y wget centos-
    release-scl devtoolset-8     devtoolset-8-libatomic-devel tcl-devel tcl tk libstdc
    ++ tk-devel pcre-devel     python36u python36u-libs python36u-devel python36u-pip
    &&     yum clean -y all &&     rm -rf /var/lib/apt/lists/*
 ---> Running in 74e7ae1d273f
Loaded plugins: fastestmirror, ovl
There is no installed groups file.
Maybe run: yum groups mark convert (see man yum)
```

```
Determining fastest mirrors
 * base: mirrors.aliyun.com
 * extras: mirrors.163.com
 * updates: mirrors.aliyun.com
Resolving Dependencies
...
Complete!
Loaded plugins: fastestmirror, ovl
Cannot open: https://centos7.iuscommunity.org/ius-release.rpm. Skipping.
Error: Nothing to do
The command '/bin/sh -c yum group install -y "Development Tools"     && yum install -y
     https://centos7.iuscommunity.org/ius-release.rpm     && yum install -y wget centos
    -release-scl devtoolset-8     devtoolset-8-libatomic-devel tcl-devel tcl tk libstdc
    ++ tk-devel pcre-devel     python36u python36u-libs python36u-devel python36u-pip
    &&     yum clean -y all &&     rm -rf /var/lib/apt/lists/*' returned a non-zero
    code: 1
$
```

当 docker 执行异常退出之后，用 `docker images` 命令，可以观察具体新产生出的 docker 镜像如下。其中有两个 <none> 的临时 docker 镜像，显然是没有完成的临时镜像。

```
$ docker images
REPOSITORY            TAG      IMAGE ID            CREATED              SIZE
<none>                <none>   8d8303883606         About an hour ago   203MB
openroad/tritonroute  latest   665ec476c7f7         About an hour ago   331MB
<none>                <none>   8405a37610b9         About an hour ago   1.99GB
openroad/yosys        latest   5dfde399a3d1         3 hours ago         3.1GB
centos                centos7  b5b4d78bc90c         2 weeks ago         203MB
```

总之在 4 月初的时候，试了几次 docker 无果后，考虑到 Ubuntu 和 CentOS“师出同门”，便有了冒险本地编译安装 OpenRoad 工具链的打算。这确实比较冒险，搞不好就把系统搞崩溃了（我裸机上运行的就是 Ubuntu 系统，而且 Ubuntu 是我的主要工作机；如果系统崩溃是一件很严重的事情）。事实上，为了安装 boost，我确实把 gnome 桌面 GUI 给弄崩溃了，着实吓了我一跳，幸好通过系统启动后的命令行能够重新安装 gnome 桌面，最后恢复原状。

但是正所谓风险越大，回报越大。在本地编译的好处是对工具链的使用非常地灵活，比如不受 docker 容器对内存大小的限制，能够全速利用计算资源。下面我们就来讲讲怎么在本地搞定 OpenRoad 工具链。

首先所需要的软件包大多都可以直接通过如下 `apt-get install` 命令安装。

```
$ sudo apt-get install build-essential clang bison flex \
libreadline-dev gawk tcl-dev libffi-dev git \
graphviz xdot pkg-config python3 libboost-system-dev \
libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

但是在具体安装的时候，还是会因为从 CentOS 到 Ubuntu 的迁移，工具链的编译还是出现 “水土不服” 的情况。影响比较深的有如下四个：

Remark: 下面提及的各个问题在具体编译时出现的的次序并不一定是这里所描述的先后次序，可以根据自己编译是出现的问题对号入座；另外，我在编译安装 OpenRoad 的时候并没有实时记录出现的问题以及我当时是怎么解决的，而是通过事后的回忆。所以可能有些问题，我很容易就解决了，在事后没有留下多大的印象，也就没有回忆起来。最后，我编译时候遇到什么问题自己解决不了，都会 google 搜索一下，往往都能给出很不错的解答，也希望读者也能够靠自己的信息检索能力，自食其力，努力解决问题。从我 firefox 的搜索记录来看，出现了很多次链接的时候找不到库文件的情况，有如下报错（不一定真实出现，只是出现问题后搜到了，但肯定有关）。但是这些链接库找不到的问题往往是和没有安装要求版本的软件如 boost 和 cmake 并发出现的。所以应该优先解决软件包安装的问题，再看看有没有编译链接时找不到库的情况。

```
/usr/bin/ld: cannot find -ltcl
/usr/bin/ld: cannot find -lpq
/usr/bin/ld: cannot find -lpthreads
/usr/bin/ld: cannot find -lglut32, -lopengl32, -lglu32, -lfreegut
```

1. **CMake 程序包** `CMake 3.14...3.16 or higher is required. You are running version 3.10.2`. 在 Ubuntu 上本地编译通常都会遇到这种情况，因为通过 `apt-get install` 的方式，能够得到最高的版本是 3.10.2，并不满足编译 OpenRoad 工具链的要求。这时就需要下载源码，手动编译安装。这里可以参考链接 How to install CMake.

2. **Boost 程序包** `Detected version of Boost is too old. Requested version was 1.68`这种情况也是在 Ubuntu 系统中通常会遇到的情况。因为通过 `apt-get install` 的方式，能够得到最高的版本是 1.65。这里需要更新一下 `apt-get` 的 system's software sources，也即 **Install Boost C++ libraries From PPA** 的方式。然后再 `apt-get install`

```
$ sudo add-apt-repository ppa:mhier/libboost-latest
$ sudo apt update
$ sudo apt install libboost1.68-dev
```

3. **tcl.h 头文件** 在老师发的任务手册中指出："编译有错误，本文的实验机器上，abc 有个文件需把 `tcl.h` 改为 `tcl/tcl.h`". 我自己编译的时候，发现这段叙述中有三个问题。第一，abc 不只有一个文件需要求改 tcl 相关的头文件，而是很多个；第二，不仅要修改 `tcl.h` 头文件的地址，还有两个类似的头文件 include 的时候需要修改地址（稍后会介绍）；第三，abc 中的这些文件都是需要先编译才会产生的，没有编译时是找不到的（一开始我找了很久都没有找到 >_<!）。这样一个个文件修改的方法就太低效了。为此我选择使用软链接的方式，一劳永逸！

```
$ cd /usr/include
$ ln -s tcl8.5/tcl.h tcl.h
$ ln -s tcl8.5/tclDecls.h tclDecls.h
$ ln -s tcl8.5/tclPlatDecls.h tclPlatDecls.h
```

4. **KLayout 程序包** 本地安装还需要另外自己下载 KLayout 程序包源码，然后自己编译安装，可以参看链接 `https://www.klayout.de/build.html`. 我自己是将 KLayout 安装在了 OpenRoad 的根目录下，如果读者要修改 KLayout 安装的位置，请修改后面展示的 `setup_env.sh` 脚本文件。这里还需要注意：编译 Klayout 还需要安装 ruby (>= 2.5) 和 Qt (>= 4.7) 软件。

最后展示一下我自己修改的 OpenRoad 本地编译安装的脚本 `build_openroad.sh` 和设置环境变量的脚本 `setup_env.sh`

```
#!/bin/bash
# This script builds the OpenROAD tools locally

# Exit on first error
set -e

echo "INFO: using local build method. This will create binaries at tools/build/"
build_method="LOCAL"

# Clone repositories
git submodule update --init --recursive

# Local build
mkdir -p tools/build/yosys
(cd tools/yosys && make install -j$(nproc) PREFIX=../build/yosys CONFIG=gcc
    TCL_VERSION=tcl8.5)
echo "INFO: finish yosys build"

mkdir -p tools/build/TritonRoute
(cd tools/build/TritonRoute && cmake ../../TritonRoute && make -j$(nproc))
echo "INFO: finish TritonRoute build"

mkdir -p tools/build/OpenROAD
(cd tools/build/OpenROAD && cmake ../../OpenROAD && make -j$(nproc))
echo "INFO: finish OpenROAD build"
```

Listing 1: `build_openroad.sh`

```
modroot="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )/tools"

if [ ! -d "${modroot}" ]; then
  echo "Module path does not exist: ${modroot}"
  exit -1
fi

export OPENROAD=${modroot}/OpenROAD
echo "OPENROAD: ${OPENROAD}"

```

```
path_string=${modroot}/build/OpenROAD/src:${modroot}/build/TritonRoute:${modroot}/
    build/yosys/bin:${modroot}/klayout-0.26.4/bin-release

if [ -n "${PATH}" ]; then
  export PATH=$path_string:$PATH
else
  export PATH=$path_string
fi

if [ -n "${MANPATH}" ]; then
  export MANPATH=${modroot}/share/man:$MANPATH
else
  export MANPATH=${modroot}/share/man
fi

library_string=${modroot}/klayout-0.26.4/bin-release

if [ -n "${LD_LIBRARY_PATH}" ]; then
  export LD_LIBRARY_PATH=$library_string:$LD_LIBRARY_PATH
else
  export LD_LIBRARY_PATH=$library_string
fi
```

Listing 2: `setup_env.sh`

最后，OpenRoad 工具链的本地编译大功告成！真的是折腾啊！

2 RISC-V 处理器的前端 RTL 设计

这里主要采用的是我毕业设计三款处理器：CHIWEN、FUXI、BIAN[周 19]. 其中 CHIWEN 是单发射静态五级流水处理器；FUXI 是双发射静态五级流水处理器；BIAN 是双发射乱序处理器（其中最长路径为 8 级流水）。三款处理器支持 RV32I 指令集；支持中断例外处理，同时也支持总线如 AXI 总线。但是还没有设计 MMU（最主要是 RISCV 的缺页处理机制和 MIPS 不一样，我还比较陌生），但是可以配置 L1 的 Icache 和 Dcache。可以作为嵌入式处理器内存使用（但是仍需要全面的调试）。

三款处理器都是用 Chisel 语言设计开发的。我为什么会选择采用 Chisel 呢？除了尝鲜之外，在论文 [周 19] 更为详细的阐述，而这里只提纲挈领地讲一下。如果熟悉 Verilog 的读者应该明白：Verilog 的抽象等级太低，细节太多，导致描述电路的能力先天不足。总结起来，主要体现在如下七点 [周 19]：

1. 语句的逻辑集成度（或者称为密度）不高，时常一个简单的逻辑代码量却不少。
2. 变量名、函数名、参数名的管理机制原始落后，甚至都没有像 C 语言的 `struct` 结构体这种聚合化的设计。代码量大时，各个符号名称的关联往往复杂而难以管理。
3. 常用的简单逻辑代码复用度不高，代码复用基本上只能用 `module` 这样一种单一的写法，容易冗余。
4. 对多维的向量数组描述和聚合化操作支持不足。
5. 没有成熟的类型系统，无论是 `Reg` 还是 `Wire` 都是对物理信号的刻画描述，粒度太小。
6. 代码中容易出现组合环，而且现有的编译器很难跟踪。
7. 要实例化不同参数的同一模块，只能一个个实例，做不到参数的数组化。

而我毕业设计的最终目标是设计一款乱序双发射的处理器，如果用 Verilog 设计，复杂度真的不好控制。而且采用 Verilog 进行 CPU 逻辑设计，对于 CPU 系统的搭建、调试以及持续的优化、增量式演进都带来不小的挑战。所以 Chisel 就成为了一个不错的选择。一个很贴切但不是绝对准确的比喻是 Chisel 之于 Verilog，就像 JAVA 之于汇编。这个读者可以在试过 Chisel 后仔细体会一下。在 [周 19] 有详细的解释。而这里举两个在我的 CPU 设计中，真实感受到 Chisel 力量的例子 [周 19]：

1. 拷贝于毕业设计代码中的处理器核顶层文件。可以看到其中颇具特点的运算符 `<>`，在 Chisel 的术语里叫做 *Bulk Connections*，用于带输入输出端口 *Bundle*（这个概念可以理解为是 C 的 `struct` 或者 Java 中的 `class`，是一种相关联变量聚合化的设计）的电路信号之间的整体互连，大大简化了模块实例化后互相连线的逻辑。三个不同型号处理器 Core 的模块是一致的，由此可见 Chisel 强大的抽象能力。

```
class Core(implicit conf: CPUConfig) extends Module {
  val io = IO(new Bundle {
    val imem = new ImemIO(conf.xprlen)
    val dmem = new DmemIO(conf.xprlen)
  })
  val frontEnd = Module(new FrontEnd)
  val backEnd = Module(new BackEnd)
  frontEnd.io.mem <> io.imem
  backEnd.io.mem <> io.dmem
  frontEnd.io.back <> backEnd.io.front
}
```

Listing 3: Core Module

2. 拷贝于毕业设计 BIAN 处理器代码中的写回结果总线监听逻辑。发射队列有 nEntry 项，每一项存有一条待发射的指令，需要监听两个源操作数，写回总线结果对应于代码中的 *io.bypass*。下面短短几行的代码里，涵盖了将写回总线的寄存器地址和有效使能信号逐一与发射队列里的每一项待发射指令的每一个源操作数进行比对，判断是否成功监听的逻辑。Chisel 对于数组聚合化操作的强大描述能力可见一斑。

```
for (i <- 0 until nEntry) {
  for (j <- 0 until 2) {
    inst_ctrl.snoop(i)(j) := issue.snoop(i)(j).valid ||
    io.bypass.map(b =>
    b.addr === issue.snoop(i)(j).addr && b.valid).reduce(_||_)
  }
}
```

Listing 4: registers write back bus bypass logic

其实在 Listing 3 中已经“暴露”出了我的处理器设计架构，也即将整个 CPU 分为 FrontEnd 和 BackEnd 两大模块。FrontEnd 负责取指令，包括处理和外部指令界面端口的交互，以及和 icache 和取指部分 MMU（如果有的话）的交互；BackEnd 负责指令的执行写回寄存器堆，中断例外检测和处理，数据访存，包括处理和外部数据界面端口的交互，以及和 dcache 和访存部分 MMU（如果有的话）的交互。这样设计的好处在于只要定义好清晰完备的 FrontEnd 和 BackEnd 之间交互的接口，两大模块就可以独立设计、独立优化、独立迭代更新。这里不妨以 BIAN 处理器为例，展示一下带有转移猜测技术的 FrontEnd 框图1。

具体来看，CHIWEN 处理器采用的是经典的五级流水设计，各个流水级的解释如下:

- 第 1 级：**if/pc 级、取指级** 发出 PC 从内存或者指令高速缓存中取指，和经典的五级流水线保持一致。
- 第 2 级：**inst back & dec 级、指令取回级** 指令在这一级被取回然后译码得到处理器内部操作微码。

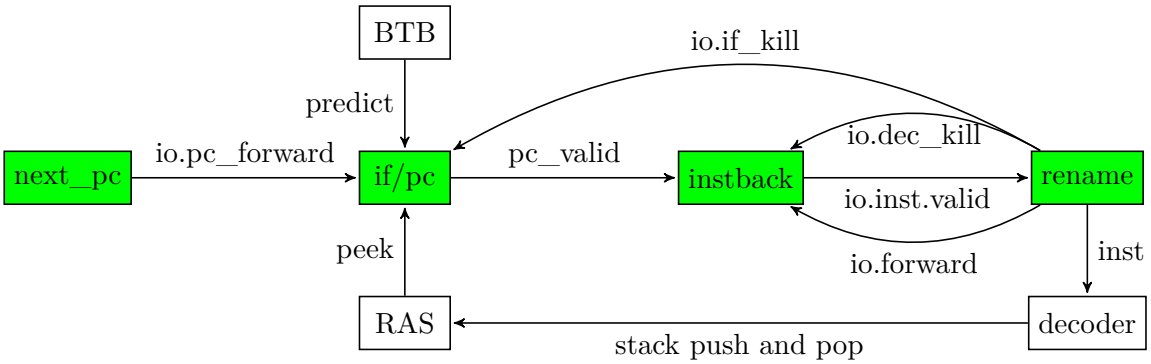


图 1: 处理器 BIAN 的 FrontEnd 流水线。([周 19] 图 3.9)

- 第 3 级：**execute 级、执行级** 主要负责 ALU 指令的执行；但是为了效率的考虑，这一级也负责发送访存请求。
- 第 4 级：**mem 级、访存级** 这一级虽然名为访存级，但只负责接收 load 指令返回的数据；中断例外也在这一级集中处理。
- 第 5 级：**write back 级、写回级** 这一级向寄存器写回处理器的值。

FUXI 处理器和 CHIWEN 处理器类似，也是这五级流水的设计。只不过 FrontEnd 可以一个周期发送两条指令给 BackEnd 执行；对应地，BackEnd 流水线每级可以容纳两条指令，可以形象地成为 “双车道”。FUXI 处理器相比 CHIWEN 处理器设计的复杂度也可以从 FrontEnd 和 BackEnd 两个角度来看。FrontEnd 中，维护状态机的正确的状态切换变得更加复杂；BackEnd 中，需要解决更多的指令之间的相关冲突。

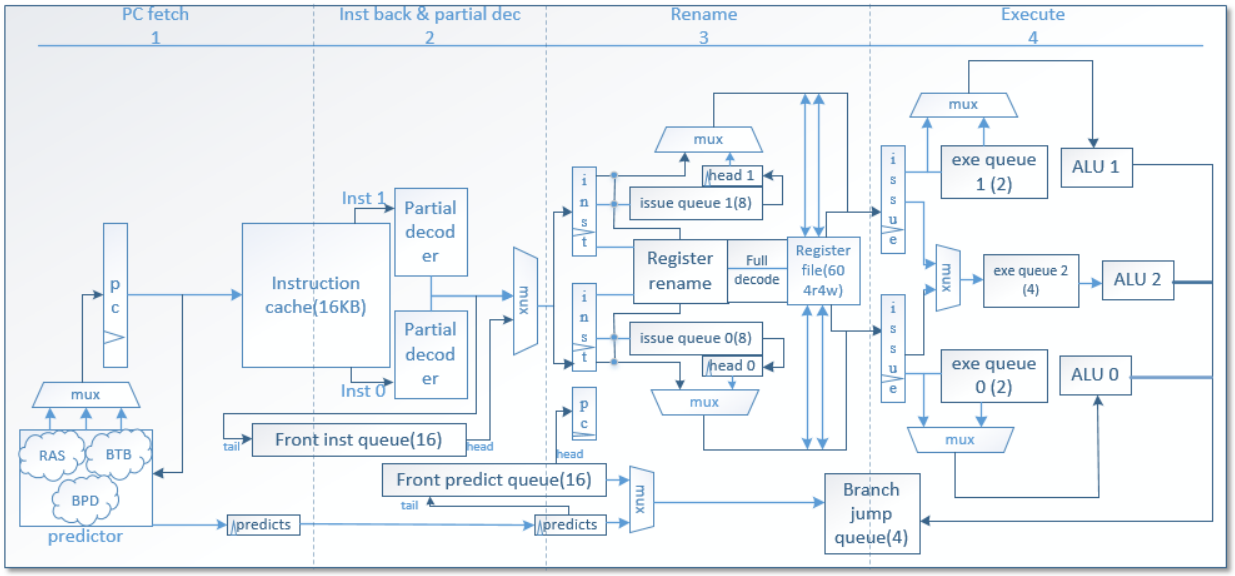


图 2: BIAN 处理器整体框图，不包括访存单元。([周 19] 图 3.2)

最后就是我本科毕业设计的大 boss：BIAN 处理器。BIAN 处理器就变得很复杂了，我做了较多的优化。而且还有很多需要修缮重构优化的地方，无论是主频、功耗还是指令执行效率 IPC。关于 BIAN 处理器的具体的各个模块的设计可以参考论文 [周 19]，我这里不想赘述，仅仅用一张整体框图2来大致展示一下。

这三款处理器首先肯定是能够通过基本的 RV32I 指令集上的 isa 测试集的。其次还能够正确执行如表1所示的八个嵌入式基准测试程序。

表 1: 嵌入式基准测试程序。除了 coremark 是自己添加的，其余的均为riscv-tests所提供的。表中最右边 3 列集成了三款处理器执行各个程序的每周期指令数（IPC）。

名称	功能解释	动态执行指令数	CHIWEN	FUXI	BIAN
coremark	A synthetic embedded integer benchmark	~ 780K	0.860	1.122	1.076
dhrystone	A synthetic embedded integer benchmark.	~ 370K	0.946	1.143	1.438
median	Performs a 1D three element median filter.	~ 15K	0.829	0.982	0.909
multiply	A software implementation of multiply.	~ 49K	0.913	1.331	1.270
qsort	Sorts an array of integers using quick sort.	~ 230K	0.780	0.866	1.046
rsort	Sorts an array of integers using radix sort.	~ 370K	0.993	1.338	1.814
towers	Solves the Towers of Hanoi puzzle recursively.	~ 18K	0.884	1.097	1.238
vvadd	Sums two arrays and writes into a third array.	~ 11K	0.853	1.078	1.223

在本节的最后展示一下这三款处理器在 [周 19] 中介绍的仿真平台上执行指令执行的效率，也即 IPC 的情况，如表1和图3所示。

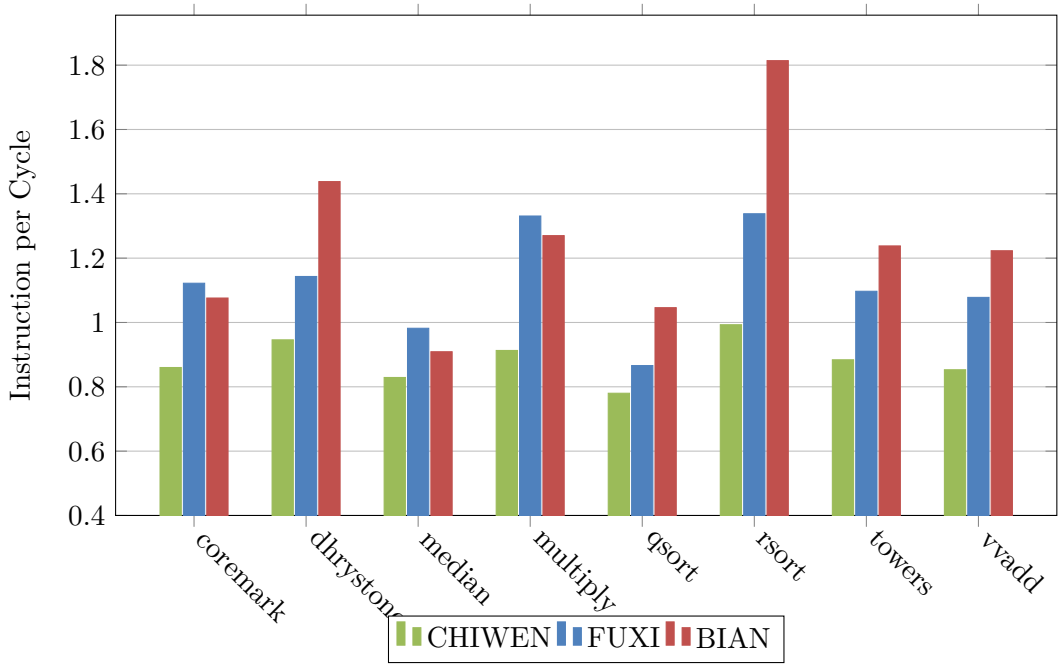


图 3: 三款处理器执行各个程序的每周期指令数（IPC）。([周 19] 图 4.6)

3 处理器 project 以及行为仿真说明

压缩包中是由我提供的本课程该大作业的 project，包括了：RTL 源码，行为仿真的 demo，和已经在 OpenRoad 工具链进行过综合、后端物理设计所需要的文件。查看 project 的目录的命令如下：

```
$ tar -zxvf project.tar.gz
$ tree demo
$ tree design
```

这样 `demo` 和 `design` 目录中的层级结构就一览无余了，见 Listing 5 和 Listing 6.

```
demo/
    README.md
    rv32_toy
        ALU.v BackEnd.v BTB.v Core.v CSRFile.v FetchInst.v
        FrontEnd.v InstDecoder.v RAS.v Regfile.v
    software
        firmware
            firmware.h
            ...
            stats.c
        Makefile
        tests
            addi.S
            ...
    src
        ALU.scala BackEnd.scala BTB.scala Core.scala CSR.scala FetchInst.scala
        FrontEnd.scala InstDecoder.scala RAS.scala Regfile.scala
    testbench
      firmware.hex Makefile testbench.v tracer.py
```

Listing 5: demo directory structure

解压缩 `project.tar.gz` 会出来两个子目录 `design` 和 `demo`。`demo` 目录中的工程文件是用来给读者自行运行行为仿真亲自体验处理器运行效果的。具体如何运行，以及需要安装什么工具链，可以参见 `demo` 下面的 `README.md` 文件。由于从 [周 19] 测试平台上将处理器核以及测试程序移植出来——能够让读者自行仿真观察处理器行为——需要一些功夫，目前 `demo` 中我只移植了 CHIWEN 一个处理器核，并重新命名为 `rv32_toy`。Verilog 的 RTL 源代码可以在目录 `demo/rv32_toy` 中找到，一共有 10 个模块。分别为：

1. **ALU** 算数逻辑预算单元，支持 RV32I 指令集的运算执行。
2. **BackEnd** BackEnd 模块，整合了 BackEnd 的所有逻辑（实例化了所有在 BackEnd 中的模块，并且 handle 它们之间的相互交互）。
3. **BTB** BTB 转移预测单元模块。

4. **Core** 处理器顶层模块。实例化了 FrontEnd 模块和 BackEnd 模块，负责 handle 这两者的交互以及分别与 CPU 外部取值端口和访存端口的交互。
5. **CSRFile** CSR 寄存器堆模块。
6. **FetchInst** 取值单元模块。
7. **FrontEnd** FrontEnd 模块，整合了 FrontEnd 的所有逻辑（实例化了所有在 FrontEnd 中的模块，并且 handle 它们之间的相互交互）。
8. **InstDecoder** 译码单元模块。
9. **RAS** RAS 转移预测单元。
10. **Regfile** 通用寄存器堆。

这 10 个模块的组织形式为：

```
Top module:  \rv32_toy
Used module:     \BackEnd
Used module:         \ALU
Used module:         \Regfile
Used module:         \InstDecoder
Used module:         \CSRFile
Used module:     \FrontEnd
Used module:         \RAS
Used module:         \BTB
Used module:         \FetchInst
```

Scala 的 Chisel 源代码可以在目录 `demo/src` 里面找到。其他的两个目录：`demo/software` 里面放置的是仿真所需要的软件；`demo/testbench` 里面放置的是仿真相关的 testbench 文件。另外，目前我还正在从 [周 19] 测试平台上将 FIXI 处理器核移植出来，并重新命名为 `rv32_ttoy`。但是从我毕设期间最后一次开发维护 CHIWEN、FUXI 和 BIAN 处理器到现在的移植已经整整过去一年的时间了，已经荒废了很久，变得“年久失修”了。就拿 `rv32_ttoy` 这个正在移植的处理器来说，虽然能够跑仿真了，但是目前跑测试程序还有 BUG（调试的效率很慢），所以我就没有把它放到 `demo` 工程里面。虽然 `rv32_ttoy` 尚有 BUG，但其实已经不妨碍用 OpenRoad 工具链进行综合和后端物理设计了，所以在 Listing 6 的第 13、14 行。

```
design/
    nangate45
        bian.mk chiwen.mk fuxi.mk rv32_toy.mk rv32_ttoy.mk
    src
        bian
            design.sdc Top.v
        chiwen
            design.sdc Top.v
```

```
        fuxi
            design.sdc Top.v
        rv32_toy
            design.sdc Top.v
        rv32_ttoy
            design.sdc Top.v
```

Listing 6: desgin directory structure

值得一提的是，这里所谓的 “移植” 还包括了 CPU 代码逻辑的简化。`rv32_toy` 和 `rv32_ttoy` 分别是 CHIWEN 和 FUXI 的简化。因为我毕设的时候论文唯一的指标就是处理器执行指令的高效性，也即为了工作的 “漂亮”，IPC 越高越好。所以我就做了很多的优化，而这在我目前看来是过度优化的。因为很多优化细节我现在印象已经模糊了。这显然不是一件好事，因为对下面章节4所论述的 OpenRoad 的综合和后端物理设计而言，最好是要清楚原始代码设计中的每一个细节，才能够在有调试 EDA 需求的时候，清楚 EDA 工具链操作流程每一步转化后对应的原始逻辑是什么样子的。所以在 EDA 的设计和调试中，我认为对原始的逻辑做到心中有数是尤其重要的。事实上，通过真实的运行，我确实已经锁定了一个 OpenRoad 工具链可能存在 BUG 的存疑地方，急需要搞清楚 OpenRoad 内部实现的细节。另外还有 3 点因素促使我必须要做简化：

1. 我个人并不想吃老本。个人想借着这次的 EDA 工具所提供的其他两个额外指标——延迟（Setup time）和功耗，重新从简到繁设计 CPU。

2. 每次过了一段时间重新看自己的代码，都有种觉得之前设计的不太优雅的地方，便有种重新修改的冲动。比如我现在看我毕设设计的取值状态机逻辑，就觉得不够优美，所以最近就在倒腾着重新采用新的状态机来重新设计取值器的逻辑。但是阻力重重，翻过一个 BUG 又是一个 BUG。

3. 现阶段的目的并不是单纯追求高性能，而是追求简洁，而从简洁的角度再循序渐进最求性能的提高。因为我毕设只是进行了行为仿真，但是 OpenRoad 工具链还能够暴露出另外两个重要的指标：CPU 延迟和功耗。CPU 简洁设计的强大就在于它往往是低延迟和低功耗的。

4 OpenRoad 的综合、后端物理设计

在上面的章节3中，我提到其实去年迫于做毕设时间上的压力（我设计这三款处理器的总时间其实只有两个多月），所以只做了指令执行效率的考虑，设计和分析，完全没有量化的延迟和功耗的考虑（本来是可以用龙芯里面的商业 EDA 工具来运行看看结果的，但是其实也没有多大意义）。我甚至连 FPGA 的 Vivado 综合都没有做，因为要在 FPGA 上跑起来，外设也要连接起来。这些外设的控制模块也不是没有，我龙芯杯的工程里有一整套，但是是基于 MIPS 的。移植到 RISC-V 也不是不行，但是怎么的都需要时间呐，上文已经提到了，去年毕设“留给中国队的时间不多了”。今年恰好有这么一个开源的 EDA 工具链，确实是一个不错的重新温顾自己处理器设计的机会，也希望能够也是一个开启自己 EDA 工具链设计之旅的契机吧。

下面就以 `demo` 中的 `rv32_toy` 处理器核为例，不过这里要从 `demo` 目录中移步到 `design` 目录中，因为 `design` 目录中存放的文件才是 OpenRoad EDA 工具链所需要的。`design` 目录按照 OpenRoad-flow 开源项目原有的组织形式又分成了 `src` 和 `nangate45` 子目录，具体运行 EDA 综合和后端物理设计之前，将这两个目录合并到 `OpenRoad-flow/flow/design` 目录中去即可。`src` 中存放的是`.sdc` 时序约束文件和`.v` 的处理器 RTL Verilog 代码（为了方便起见，我自行设计的处理器的 RTL 代码的所有模块统一集成为单个 `Top.v` 文件，而事实上这是 Chisel 编译器自动编译的，`demo` 中我故意将每个模块拆开来，是为了让读者对于各个模块能够看的更清楚）。`.sdc` 时序约束文件非常简单，定义时钟周期即可，所以基本上一两行搞定。这里我们提供的名为 `design.sdc` 的约束文件如 Listing 7 所示（和 OpenRoad-flow 开源项目自带的 tinyRocket 保持一致）。

```
set_units -time ns
create_clock [get_ports clock]  -name core_clock  -period 5.6
```

Listing 7: design.sdc

这里设置的时钟周期为 10ns，在具体的 RTL 代码中，时钟的端口名为 clock，这个一定不能搞错，因为 OpenRoad 如果在源码中找不到 clock 时钟端口，就会报错。`nangate45` 放置的是`.mk` 文件。这是一种 OpenRoad 自己定义的文件格式（mk 取自 Makefile 的缩写）。具体内容如 Listing 8 所示。其中，第一行 `DESIGN_NAME` 指的是 RTL 代码底层的模块名称；第二行 `PLATFORM`，本次报告仅采用 nangate45；第四行 `VERILOG_FILES` 对应于 RTL 源码，我们这里已经将所有源码集成为了一个 `Top.v` 文件；第五行 `SDC_FILE` 就是之前提到的约束文件；后面依次是设置 `DIE_AREA`, `CORE_AREA`, `CLOCK_PERIOD` 变量（和 tinyRocket 的设置相同），这些变量都顾名思义，很好理解。

```
export DESIGN_NAME = rv32_toy
export PLATFORM    = nangate45

export VERILOG_FILES = ./designs/src/rv32_toy/Top.v
export SDC_FILE      = ./designs/src/rv32_toy/design.sdc

# These values must be multiples of placement site
# x=0.19 y=1.4
export DIE_AREA    = 0 0 924.92 799.4
```

```
10 export CORE_AREA    = 10.07 9.8 914.85 789.6
11
12 export CLOCK_PERIOD = 5.600
```

Listing 8: rv32_toy.mk

有了 RTL 文件，时序约束文件和 mk 文件，就可以运行 OpenRoad 工具链了。如下为执行的命令：

```
1 # At OpenRoad-flow top directory
2 $ source setup_env.sh
3 $ cd flow
4 $ make DESIGN_CONFIG=./designs/nangate45/rv32_toy.mk
```

首先我们需要关注的是集成在 OpenROAD-flow/flow 目录的 Makefile 中定义的涉及到 OpenRoad 工具链中各个工具使用的整个 flow 流程。而认识整个流程的重要依据就是工具链运行结束后，在 flow 目录的子目录 logs，objects，reports，results 产生的和 rv32_toy 处理器相关的文件。除了 objects 子目录我们不是很关心以外，其他三个子目录我们依次来看。

4.1 log 目录与执行流程

首先是 logs 子目录，里面存放的是 make 命令回车之后，工具链调用各个子工具对 Verilog 编写的处理器 RTL 代码进行分析转化的过程中，Terminal 上打印出来的实时的记录。所以这些记录对于明确整个综合布局布线过程的流程和出现错误的事后复盘锁定产生错误的原因来说，都是非常重要的信息。不难发现 logs 中的记录文件的命名和着 flow/README.md 中的图4展示的步骤是一致的。整个工具链输入为 Design（Verilog 编写的 RTL 文件.v、SDC 时序约束文件.sdc）和 Tech（liberty 文件.lib 和 technology 文件.lef）最后输出的是 GDS 的版图（generate tapeout-ready GDSII）文件。在输入和输出之间被划分成了 6 步，下面我们结合着目录 flow/logs 中的记录、flow/Makefile 中的编译步骤和目录 flow/scripts 中的脚本文件（脚本中的 tcl 命令可以参考网页https://openroad.readthedocs.io/en/latest/user/user-guide.html#option-2-individual-flow-steps上的说明）来介绍：

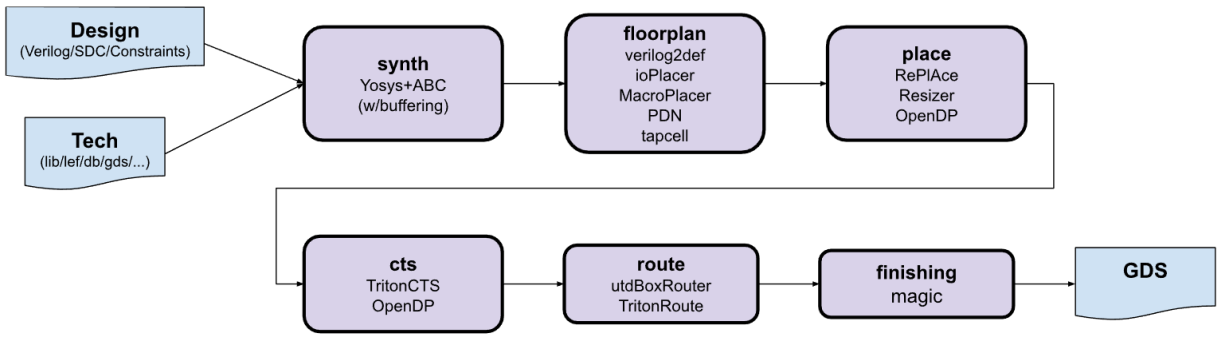


图 4: OpenRoad 工具链的输入输出、流程以及流程中每个步骤所使用到的工具。

1. **logic synthesis (LS)** 用到的工具是 Yosys+ABC。执行脚本文件 `synth.tcl`，生成了 log 文件 `1_1_yosys.log`. 传统意义上 Yosys 负责 LS 的前半段，ABC 负责 LS 的后半段。从 log 上看也确实如此。log 中显示一共有 18 个环节，真正开始执行 ABC 是从 10.1.1 环节开始的，如下：

```
10.1.1. Executing ABC.
Running ABC command: OpenROAD-flow/tools/build/yosys/bin/yosys-abc -s -f /tmp/
    yosys-abc-OhaHX2/abc.script 2>&1
ABC: ABC command line: "source /tmp/yosys-abc-OhaHX2/abc.script".
ABC:
ABC: + read_blif /tmp/yosys-abc-OhaHX2/input.blif
ABC: + read_lib -w OpenROAD-flow/flow/./objects/nangate45/rv32_toy/merged.lib
```

但是在 OpenRoad 里面其实已经突破了这种局限性，Yosys 已经把 ABC 和 RePlAce 工具集成在了一起。做了两点改进 [ACF+19]：首先，运用了增强学习来使能为了时序驱动的逻辑优化的自动化设计空间搜索；并且开发了一个增强学习代理来自动产生 step-by-step 的综合脚本使得在满足时序约束的情况下最小化面积；其次，提升了在 ABC 中基础的 buffering 算法，并且通过集成 RePlAce 工具使能了物理感知 buffering 和门大小；凭借全局的基于布局的线路电容估计来在门大小和 buffering 上来提高综合后的时序结果。整个过程包括分析代码的层次结构，并根据源码生成 AST 表示；根据 Cell Library 将 RTL 代码转化为门级网表。这一转化的内容很多，同时这里也会像编译器一样执行各个优化的 pass，而且同样的 pass 会在不同的优化迭代和不同的优化组合中出现多次。统计出现的各种 pass 如下，一共 40 种：
`PROC_CLEAN` pass (remove empty switches from decision trees), `PROC_RMDEAD` pass (remove dead branches from decision trees), `PROC_PRUNE` pass (remove redundant assignments in processes), `PROC_INIT` pass (extract init attributes), `PROC_ARST` pass (detect async resets in processes), `PROC_MUX` pass (convert decision trees to multiplexers), `PROC_DLATCH` pass (convert process syncs to latches), `PROC_DFF` pass (convert process syncs to FFs), `FLATTEN` pass (flatten design), `OPT_EXPR` pass (perform const folding), `OPT_CLEAN` pass (remove unused cells and wires), `CHECK` pass (checking for obvious problems), `OPT_MERGE` pass (detect identical cells), `OPT_MUXTREE` pass (detect dead branches in mux trees), `OPT_REDUCE` pass (consolidate $*mux and $reduce_* inputs), `OPT_RMDFF` pass (remove dff with constant values), `WREDUCE` pass (reducing word size of cells), `PEEPOPT` pass (run peephole optimizers), `TECHMAP` pass (map to technology primitives), `ALUMACC` pass (create $alu and $macc cells), `SHARE` pass (SAT-based resource sharing), **`FSM pass`** (extract and optimize FSM), `FSM_DETECT` pass (finding FSMs in design), `FSM_EXTRACT` pass (extracting FSM from design), `FSM_OPT` pass (simple optimizations of FSMs), `FSM_RECODE` pass (re-assigning FSM state encoding), `FSM_INFO` pass (dumping all available information on FSM cells), **`FSM_MAP` pass (mapping FSMs to basic logic), `OPT_MEM` pass (optimize memories), `MEMORY_DFF` pass (merging $dff cells to $memrd and $memwr), `MEMORY_SHARE` pass (consolidating $memrd/$memwr**

cells), **MEMORY_COLLECT pass (generating $mem cells)**, **MEMORY_MAP pass (converting $mem cells to logic and flip-flops)**, **OPT_SHARE** pass, **ABC** pass (technology mapping using ABC), **DFFLIBMAP** pass (mapping DFF cells to sequential cells from liberty file), **HILOMAP** pass (mapping to constant drivers), **SETUNDEF** pass (replace undef values with defined constants), **SPLITNETS** pass (splitting up multi-bit signals), **INSBUF** pass (insert buffer cells for connected wires). 最后给出所占各项资源的统计，如下所示：

```
17. Printing statistics.
=== rv32_toy ===
Number of wires:                 76729
Number of wire bits:             76886 # 后面步骤会反复用到，也即Nets的数量
Number of public wires:           8703
Number of public wire bits:       8860
Number of memories:                  0
Number of memory bits:               0
Number of processes:                 0
Number of cells:                 69013 # 后面步骤会反复用到，也即components/
    PlaceInstances的数量
  # 下面是各种具体Cell的使用量，省略之
Chip area for module '\rv32_toy': 110447.722000
```

2. **floorplan (FP) and power delivery network (PDN) generation** 这一阶段又分出 6 个子步骤：

   (a) Translate verilog to def. 用到的工具是 verilog2def。执行脚本文件floorplan.tcl，生成了 log 文件 2_1_floorplan.log. 会分析得出初步的时序和面积报告，如下所示（和流程后续步骤的时序和面积分析结果有偏差）：

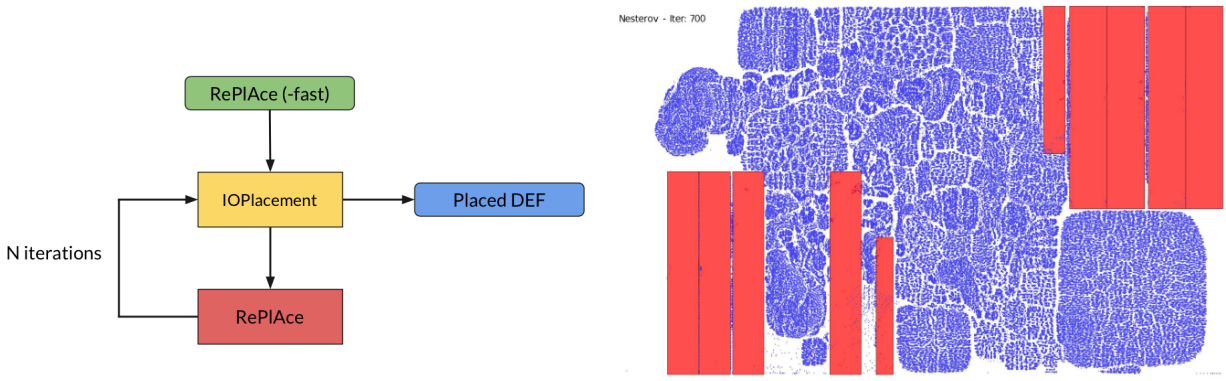
```
report_checks
Startpoint: _137035_ (rising edge-triggered flip-flop clocked by core_clock)
Endpoint: _130077_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: max
# 此处略去具体的路径
 5.60    5.60   clock core_clock (rise edge)
 0.00    5.60   clock network delay (ideal)
 0.00    5.60   clock reconvergence pessimism
         5.60 ^ _130077_/CK (DFF_X1)
-0.04    5.56   library setup time
         5.56   data required time
        -1.92   data arrival time
-----------------------------------------------------------
         3.64   slack (MET)
report_tns
tns 0.00
```

```
report_wns
wns 0.00
report_design_area
Design area 441791 u^2 16% utilization.
```

(b) IO Placement. 用到的工具是 ioPlacer。执行脚本文件`io_placement.tcl`，生成了 log 文件 `2_2_floorplan_io.log`. 进行 IO 引脚的布局，如图5a展示了这一步骤的迭代过程。log 结果如下：

```
> Running IO placement
* Num of slots          10578
* Num of I/O            170
* Num of I/O w/sink     170
* Num of I/O w/o sink   0
* Slots Per Section     200
* Slots Increase Factor 0.01
* Usage Per Section     0.8
* Usage Increase Factor 0.01
* Force Pin Spread      1
```

(c) Timing Driven Mixed Sized Placement. 用到的工具是 RePIAce。执行脚本文件`tdms_place.tcl`，生成了 log 文件为 `2_3_tdms_place.log`. 所谓的 mixed size 就是 macros 宏单元和 standard cell 标准单元的混合布局，如图5b。对于 rv32_toy 来说，因为没有宏单元，所以



(a) (b)

图 5: (a) ioPlacer, (b) Foundry 16nm RISC-V based design block from the University of Michigan, after RePlAce mixed-size placement. Red color indicates macros and blue color indicates standard cells. ([ABC+19] 图 4)

这一步跳过了。

```
No macros found: Skipping global_placement
```

(d) Macro Placement. 用到的工具是 MacroPlacer。执行脚本文件`macro_place.tcl`，生成

了 log 文件为 `2_4_mplace.log`. macro-packing 使用基于 Parquet[AM03] 的模拟退火算法。但是因为 rv32_toy 没有宏单元，这一步也跳过了。

(e) Tapcell and Welltie insertion. 用到的工具是 tapcell。执行脚本文件`tapcell.tcl`，生成了 log 文件为 `2_5_tapcell.log`. 这一步负责在设计中插入 endcaps 和 tapcells。

```
Step 1: Cut rows...
---- Macro blocks found: 0
---- #Original rows: 557
---- #Cut rows: 0
Step 2: Insert endcaps...
---- #Endcaps inserted: 1114
Step 3: Insert tapcells...
---- #Tapcells inserted: 1957
```

(f) PDN generation（Power Delivery Network）. 用到的工具是 TritionFPlan/pdngen。执行脚本文件`pdn.tcl`，生成了 log 文件为 `2_6_pdn.log`. Floorplan 阶段的 PDN 基于使用 single pitch throught the chip，对应的 log 如下所示：

```
Type: stdcell, grid
Stdcell Rails
Layer: metal1 -  width: 0.170  pitch: 2.400  offset: 0.000
Straps
Layer: metal4 -  width: 0.480  pitch: 56.000  offset: 2.000
Layer: metal7 -  width: 1.400  pitch: 40.000  offset: 2.000
Connect: {metal1 metal4} {metal4 metal7}
Type: macro, macro_1
Macro orientation: R0 R180 MX MY
Straps
Layer: metal5 -  width: 0.930  pitch: 40.000  offset: 2.000
Layer: metal6 -  width: 0.930  pitch: 40.000  offset: 2.000
Connect: {metal4_PIN_ver metal5} {metal5 metal6} {metal6 metal7}
Type: macro, macro_2
Macro orientation: R90 R270 MXR90 MYR90
Straps
Layer: metal6 -  width: 0.930  pitch: 40.000  offset: 2.000
Connect: {metal4_PIN_hor metal6} {metal6 metal7}
[INFO] [PDNG-0013] Inserting stdcell grid - grid
[INFO] [PDNG-0015] Writing to database
```

3. **placement** 这一阶段又分出 3 个子步骤：

(a) Global placement. 使用的工具为 RePIAce（这一步是 RePIAce 的主要业务）。执行脚本文件`global_place.tcl`，生成了 log 文件 `3_1_place_gp.log`. 进行标准单元在给定 floorplan 后的布局。其中采用了 Nesterov 加速与适应的梯度下降算法来求解最优值[CKKW18]，反映在 log 里为：

```
Reading DEF file: ./results/nangate45/rv32_toy/2_floorplan.def
Notice 0: Design: rv32_toy
Notice 0:     Created 170 pins.
Notice 0:     Created 72084 components and 378066 component-terminals.
Notice 0:     Created 5 special nets and 144168 connections.
Notice 0:     Created 76886 nets and 233898 connections.
Notice 0: Finished DEF file: ./results/nangate45/rv32_toy/2_floorplan.def
[INFO] DBU = 2000
[INFO] SiteSize = (380, 2800)
[INFO] CoreAreaLxLy = (20140, 19600)
[INFO] CoreAreaUxUy = (1829700, 1579200)
[INFO] NumInstances = 72084
[INFO] NumPlaceInstances = 69013
[INFO] NumFixedInstances = 3071
[INFO] NumDummyInstances = 0
[INFO] NumNets = 76886
[INFO] NumPins = 234068
[INFO] DieAreaLxLy = (0, 0)
[INFO] DieAreaUxUy = (1849840, 1598800)
[INFO] CoreAreaLxLy = (20140, 19600)
[INFO] CoreAreaUxUy = (1829700, 1579200)
[INFO] CoreArea = 2822189776000
[INFO] NonPlaceInstsArea = 3267544000
[INFO] PlaceInstsArea = 441790888000
[INFO] Util(%) = 15.672334
[INFO] StdInstsArea = 441790888000
[INFO] MacroInstsArea = 0
[InitialPlace]  Iter: 1 CG Error: 0.000220941 HPWL: 783488300
# 初始化布局，使用共轭梯度算法，按照2，3，...，19步迭代，并在20步迭代后终止。
[InitialPlace]  Iter: 20 CG Error: 1.3889e-05 HPWL: 746919615
[INFO] FillerInit: NumGCells = 136924
[INFO] FillerInit: NumGNets = 76886
[INFO] FillerInit: NumGPins = 234068
[INFO] TargetDensity = 0.300000
[INFO] AveragePlaceInstArea = 6401560
[INFO] IdealBinArea = 21338532
[INFO] IdealBinCnt = 132257
[INFO] TotalBinArea = 2822189776000
[INFO] BinCnt = (256, 256)
[INFO] BinSize = (7069, 6093)
[INFO] NumBins = 65536
[NesterovSolve] Iter: 1 overflow: 0.996213 HPWL: 317581770
# 省略了中间的迭代过程，迭代的单位是10步，依次为10，20，...，490
[NesterovSolve] Iter: 500 overflow: 0.107354 HPWL: 3690870791
[NesterovSolve] Finished with Overflow: 0.0989353
```

最后的 NesterovSolve 终止的条件是 Overflow 小于 0.1。Nesterov 梯度下降的可视化可以参见附件中的 3 张 gif 动态效果图，非常直观生动。

(b) Resizing & Buffering. 用到的工具是 resizer。执行脚本文件`resize.tcl`，生成了 log 文件 `3_2_resizer.log`. resizer 在这一步进行的是 Gate Resizing and buffering。这一步还会进行第二轮的时序和面积报告，如下：

```
Reading DEF file: ./results/nangate45/rv32_toy/3_1_place_gp.def
Notice 0: Design: rv32_toy
Notice 0:     Created 170 pins.
Notice 0:     Created 72084 components and 378066 component-terminals.
Notice 0:     Created 5 special nets and 144168 connections.
Notice 0:     Created 76886 nets and 233898 connections.
Notice 0: Finished DEF file: ./results/nangate45/rv32_toy/3_1_place_gp.def
============================================================================
report_checks
----------------------------------------------------------------------------
Startpoint: _130172_ (rising edge-triggered flip-flop clocked by core_clock)
Endpoint: _130766_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: max
# 此处略去具体的路径
 5.60    5.60   clock core_clock (rise edge)
 0.00    5.60   clock network delay (ideal)
 0.00    5.60   clock reconvergence pessimism
         5.60 ^ _130766_/CK (DFF_X1)
-0.04    5.56   library setup time
         5.56   data required time
        -2.67   data arrival time
-----------------------------------------------------------
         2.89   slack (MET)
report_tns
tns 0.00
report_wns
wns 0.00
============================================================================
report_design_area
----------------------------------------------------------------------------
Design area 445058 u^2 16% utilization.
Perform port buffering...
Inserted 67 input buffers.
Inserted 102 output buffers.
Perform resizing...
Resized 25420 instances.
Repair max cap...
Found 69 max capacitance violations.
```

```
Inserted 174 buffers in 69 nets.
Repair max slew...
Found 5 max slew violations.
Inserted 4 buffers in 5 nets.
Repair tie lo fanout...
Repair tie hi fanout...
Repair max fanout...
Repair max fanout...
Inserted 0 hold buffers.
============================================================================
report_floating_nets
----------------------------------------------------------------------------
Warning: found 6481 floatiing nets.
============================================================================
report_checks
----------------------------------------------------------------------------
Startpoint: _130188_ (rising edge-triggered flip-flop clocked by core_clock)
Endpoint: _130763_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: max
# 此处略去具体的路径
 5.60    5.60   clock core_clock (rise edge)
 0.00    5.60   clock network delay (ideal)
 0.00    5.60   clock reconvergence pessimism
         5.60 ^ _130763_/CK (DFF_X1)
-0.03    5.57   library setup time
         5.57   data required time
        -2.25   data arrival time
-----------------------------------------------------------
         3.32   slack (MET)
report_tns
tns 0.00
report_wns
wns 0.00
============================================================================
report_design_area
----------------------------------------------------------------------------
Design area 484065 u^2 17% utilization.
```

(c) Detail placement. 用到的工具是 OpenDP。执行脚本文件`detail_place.tcl`，生成了 log 文件 `3_3_opendp.log`. 负责做一些更细致的布局与分析，log 如下所示：

```
Design Stats
--------------------------------
total instances         72431
multi row instances         0
```

```
fixed instances          3071
nets                    77238
design area          705547.4 u^2
fixed area              816.9 u^2
movable area         120199.3 u^2
utilization                17 %
utilization padded         30 %
rows                      557
row height                1.4 u

Placement Analysis
--------------------------------
total displacement    87718.3 u
average displacement      1.2 u
max displacement         14.2 u
original HPWL        1873485.8 u
legalized HPWL       1931730.3 u
delta HPWL                   3 %
```

4. **clock tree synthesis (CTS)** 这一阶段又分出 2 个子步骤:

   (a) Run TritonCTS. 用到的工具是 TritonCTS。执行脚本文件`cts.tcl`，生成了 log 文件为 `4_1_cts_prefillcell.log`. TritonCTS 负责时钟树综合（clock tree synthesis, CTS），以低功耗、低偏移（low-skew）和低时延的时钟分布为综合的目标；基于 GH-Tree（generalized H-Tree）范式 [HKL18]。具体的用到了动态规划算法以及 k-means 算法 [ACF+19]。这一步的 log 中也会给出时序和面积报告，但是和前两步的 Resizing & Buffering 和 Detail placement 有一定的差别。这里仅展示和时钟树 H-Tree 拓扑有关的 log 信息，如下:

```
 *****************
* TritonCTS 2.0 *
*****************
*****************************
*  Import characterization  *
*****************************
Reading LUT file "./platforms/nangate45/tritonCTS/lut.txt"
  Min. len    Max. len    Min. cap    Max. cap   Min. slew   Max. slew
  2           8           1          52          1          24
  [WARNING] 180 wires are pure wire and no slew degration.
  TritonCTS forced slew degradation on these wires.
  Num wire segments: 4994
  Num keys in characterization LUT: 1677
  Actual min input cap: 8
Reading solution list file "./platforms/nangate45/tritonCTS/sol_list.txt"
**********************
```

```
*  Find clock roots  *
**********************
User did not specify clock roots.
Using OpenSTA to find clock roots.
Looking for clock sources...
  Clock names: clock
************************
*  Populate TritonCTS  *
************************
Initializing clock nets
Number of user-input clocks: 1 ( "clock" )
Looking for clock nets in the design
Net "clock" found
clock
****************************
*  Check characterization  *
****************************
The chacterization used 1 buffer(s) types. All of them are in the loaded DB.
***********************
*  Build clock trees  *
***********************
Generating H-Tree topology for net clock...
  Tot. number of sinks: 7805
Wire segment unit: 20000 dbu (10 um)
Original sink region: [(241490, 105170), (1687010, 1496430)]
Normalized sink region: [(12.0745, 5.2585), (84.3505, 74.8215)]
  Width:  72.276
  Height: 69.563
Level 1
  Direction: Horizontal
  # sinks per sub-region: 3903
  Sub-region size: 36.138 X 69.563
  Segment length (rounded): 18
  Key: 3192 outSlew: 1 load: 1 length: 8 isBuffered: 1
  Key: 3192 outSlew: 1 load: 1 length: 8 isBuffered: 1
  Key: 34 outSlew: 3 load: 1 length: 2 isBuffered: 0
Level 2
  Direction: Vertical
  # sinks per sub-region: 1952
  Sub-region size: 36.138 X 34.7815
  Segment length (rounded): 18
  Key: 3220 outSlew: 1 load: 1 length: 8 isBuffered: 1
  Key: 3192 outSlew: 1 load: 1 length: 8 isBuffered: 1
  Key: 0 outSlew: 2 load: 1 length: 2 isBuffered: 0
Level 3
```

```
  Direction: Horizontal
  # sinks per sub-region: 976
  Sub-region size: 18.069 X 34.7815
  Segment length (rounded): 10
  Key: 3206 outSlew: 1 load: 1 length: 8 isBuffered: 1
  Key: 436 outSlew: 1 load: 1 length: 2 isBuffered: 1
Level 4
  Direction: Vertical
  # sinks per sub-region: 488
  Sub-region size: 18.069 X 17.3908
  Segment length (rounded): 8
  Key: 3192 outSlew: 1 load: 1 length: 8 isBuffered: 1
Level 5
  Direction: Horizontal
  # sinks per sub-region: 244
  Sub-region size: 9.0345 X 17.3908
  Segment length (rounded): 4
  Key: 1152 outSlew: 12 load: 1 length: 4 isBuffered: 1
Level 6
  Direction: Vertical
  # sinks per sub-region: 122
  Sub-region size: 9.0345 X 8.69538
  Segment length (rounded): 4
  Key: 1242 outSlew: 12 load: 1 length: 4 isBuffered: 1
Level 7
  Direction: Horizontal
  # sinks per sub-region: 61
  Sub-region size: 4.51725 X 8.69538
  Segment length (rounded): 2
  Key: 548 outSlew: 2 load: 1 length: 2 isBuffered: 1
Level 8
  Direction: Vertical
  # sinks per sub-region: 31
  Sub-region size: 4.51725 X 4.34769
  Segment length (rounded): 2
  Key: 0 outSlew: 2 load: 1 length: 2 isBuffered: 0
  [WARNING] Creating fake entries in the LUT.
Level 9
  Direction: Horizontal
  # sinks per sub-region: 16
  Sub-region size: 2.25863 X 4.34769
  Segment length (rounded): 1
  Key: 5030 outSlew: 12 load: 1 length: 1 isBuffered: 1
Level 10
  Direction: Vertical
```

```
  # sinks per sub-region: 8
  Sub-region size: 2.25863 X 2.17384
  Segment length (rounded): 1
  Key: 5039 outSlew: 12 load: 1 length: 1 isBuffered: 1
Stop criterion found. Max number of sinks is (15)
Building clock sub nets...
Number of sinks covered: 7805
Clock topology of net "clock" done.
```

(b) Filler insertion. 用到的是 OpenDP。执行脚本文件`fillcell.tcl`，生成了 log 文件`4_cts.log`，负责布局填充实例。

```
Placed 235632 filler instances.
```

5. **routing** 这一阶段又分出 2 个子步骤：

(a) Run global route. 用到的工具是 UTD-BoxRouter。执行脚本文件`cts.tcl`，生成了 log 文件 `5_1_fastroute.log`. UTD-BoxRouter 是 BoxRouter 2.0 的修改版。该工具为读取 LEF 和布局后的 DEF 文件。它定义了全局布线的单元 cell——即熟知的 gcells；并且会运行全局的布线最小化 cells 的拥堵和溢出，与此同时最小化连线 wire 的长度和过孔。该工具还会产生一个 `route.guide` 文件（在 `results` 目录中）来指导布线细节。具体地，全局的布线器会采用预布线，整数线性规划和基于协商的 A^* 搜索布线的稳定性等算法，来首先解决 2D 布线问题；接着运行一个 2D-to-3D 的映射——using a layer assignment using an integer linear programming algorithm that is aware of vias and blockages. 产生的 log 如 Listing 9 所示。

```
Reading DEF file: ./results/nangate45/rv32_toy/4_cts.def
Notice 0: Design: rv32_toy
Notice 0:     Created 100000 Insts
Notice 0:     Created 200000 Insts
Notice 0:     Created 300000 Insts
Notice 0:     Created 170 pins.
Notice 0:     Created 309899 components and 858062 component-terminals.
Notice 0:     Created 5 special nets and 619798 connections.
Notice 0:     Created 79069 nets and 238264 connections.
Notice 0: Finished DEF file: ./results/nangate45/rv32_toy/4_cts.def
Adjust layer 2 in 50.0%
Adjust layer 3 in 50.0%

 *****************
 *   FastRoute   *
 *****************

 > Params:
 > ---- Min routing layer: 2
```

```
> ---- Max routing layer: 10
> ---- Global adjustment: 0.15
> ---- Unidirectional routing: 1
> ---- Clock net routing: 0
> Initializing grid...
> Initializing grid... Done!
> Initializing routing layers...
> Initializing routing layers... Done!
> Initializing routing tracks...
> Initializing routing tracks... Done!
> Setting capacities...
> Setting capacities... Done!
> Setting spacings and widths...
> Setting spacings and widths... Done!
> Initializing nets...
> ----Checking pin placement...
> ----Checking pin placement... Done!
> Initializing nets... Done!
> Adjusting grid...
> Adjusting grid... Done!
> Computing track adjustments...
> Computing track adjustments... Done!
> Computing obstacles adjustments...
> ----Processing 479055 obstacles in layer 1
> ----Processing 83 obstacles in layer 4
> ----Processing 98 obstacles in layer 7
> Computing obstacles adjustments... Done!
> Computing user defined adjustments...
> Computing user defined adjustments... Done!
> Computing user defined layers adjustments...
> Computing user defined layers adjustments... Done!
> Running FastRoute...
> --Running extra iterations to remove overflow...
> ----updateType 1
> ----iteration 1, enlarge 25, costheight 6, threshold 10 via cost 2
> ----log_coef 0.549593, healingTrigger 0 cost_step 2 L 1 cost_type 1
  updatetype 1
# 次数略去了中间的迭代
> ----updateType 2
> ----iteration 19, enlarge 135, costheight 100, threshold 0 via cost 2
> ----log_coef 2.000000, healingTrigger 0 cost_step 5 L 1 cost_type 1
  updatetype 2
>
> --Final usage/overflow report:
>
```

```
> ----Total Usage   : 1249456
> ----Total Capacity: 4460960
> ----Max H Overflow: 0
> ----Max V Overflow: 0
> ----Max Overflow  : 0
> ----H   Overflow  : 0
> ----V   Overflow  : 0
> ----Final Overflow: 0
>
> --Final routing length : 1249456
> --Final number of via  : 529743
> --Final total length 1 : 1779199
> --Final total length 3 : 2838685
> --3D runtime: 81.723709 sec
> --Getting results...
> --Getting results... Done!
>
> Running FastRoute... Done!
> Writing guides...
> Num routed nets: 72588
> Writing guides... Done! # 将指导写入route.guide文件中
```

Listing 9: The log info of `5_1_fastroute.log` file

(b) Run detail route. 用到的工具是 TrionRoute[KWX18]。这里用到的工具是 TritonRoute，生成了 log 文件 `5_2_TritonRoute.log`. 这一步中和根据上一步生成的指导文件 `route.guide` 提供的全局布线方案对 signal nets 和 clock nets 进行细节上的布线。不过在细节的布线之前，TritonRoute 会做两个预处理工作 [ACF+19]：1）使用宽度优先搜索来预处理全局布线方案，来缩减 " the probability of loops generated in later stage while net connectivity is preserved"；2）识别出唯一的实例 "considering orientation and routing track offsets" 和产生引脚 "access patterns to aid connections to pins". 在这一步中还会使用迭代优化尽可能地消除违例。对于 rv32_toy 一共执行了 7 轮迭代优化，最后一轮的记录如下：

```
start 7th optimization iteration ...
    completing 10% with 33 violations
    elapsed time = 00:00:00, memory = 965.88 (MB)
    completing 20% with 33 violations
    elapsed time = 00:00:00, memory = 965.88 (MB)
    completing 30% with 33 violations
    elapsed time = 00:00:00, memory = 965.88 (MB)
    completing 40% with 33 violations
    elapsed time = 00:00:01, memory = 965.88 (MB)
    completing 50% with 33 violations
    elapsed time = 00:00:01, memory = 965.88 (MB)
```

```
    completing 60% with 33 violations
    elapsed time = 00:00:01, memory = 965.88 (MB)
    completing 70% with 33 violations
    elapsed time = 00:00:02, memory = 965.88 (MB)
    completing 80% with 33 violations
    elapsed time = 00:00:02, memory = 965.88 (MB)
    completing 90% with 33 violations
    elapsed time = 00:00:02, memory = 965.88 (MB)
    completing 100% with 33 violations
    elapsed time = 00:00:02, memory = 965.88 (MB)
  number of violations = 33
```

Listing 10: violation

最后 33 个违例一直驻留着，无法取消，也只好作罢。

6. **layout finishing** 这一步就是生成版图，产生最后报告结果的收尾步骤了。执行脚本文件final_report.tcl,生成了 log 文件 6_report.log 和 reports 目录下的文件 6_final_report.rpt 的报告信息重合，留到下面再看。

4.2 reports 目录与执行报告

该目录中主要有 5 个.rpt 文件，分别为:

1. 2_init.rpt，由脚本文件floorplan.tcl产生，对应于流程中的第 2 个步骤的第一小步。和 log 文件 2_1_floorplan.log 中的信息是一致的。

2. 3_pre_resize.rpt，由脚本文件resize.tcl产生，对应于流程中的第 3 个步骤的第二小步，是 resize 调整之前的报告。和 log 文件 3_2_resizer.log 前半段的信息是一致的。

3. 3_post_resize.rpt，由脚本文件resize.tcl产生，对应于流程中的第 3 个步骤的第二小步，是 resize 调整之后的报告。和 log 文件 3_2_resizer.log 后半段的信息是一致的。

4. 5_route_drc.rpt，由 Makefile 产生，对应于流程中的第 5 个步骤的第二小步。报告里面记录了 Listing 10 的 33 个违例的具体信息。

5. 6_final_report.rpt，由脚本文件final_report.tcl产生，对应于流程中的最后一个步骤。也是对整个 RTL 设计最重要的一个报告文件。如 Listing 11 所示，就是出于篇幅考虑而删减过（保留了核心信息）的最终报告。值得一提的是，这些时序和功耗的报告是由工具 OpenSTA 产生的。

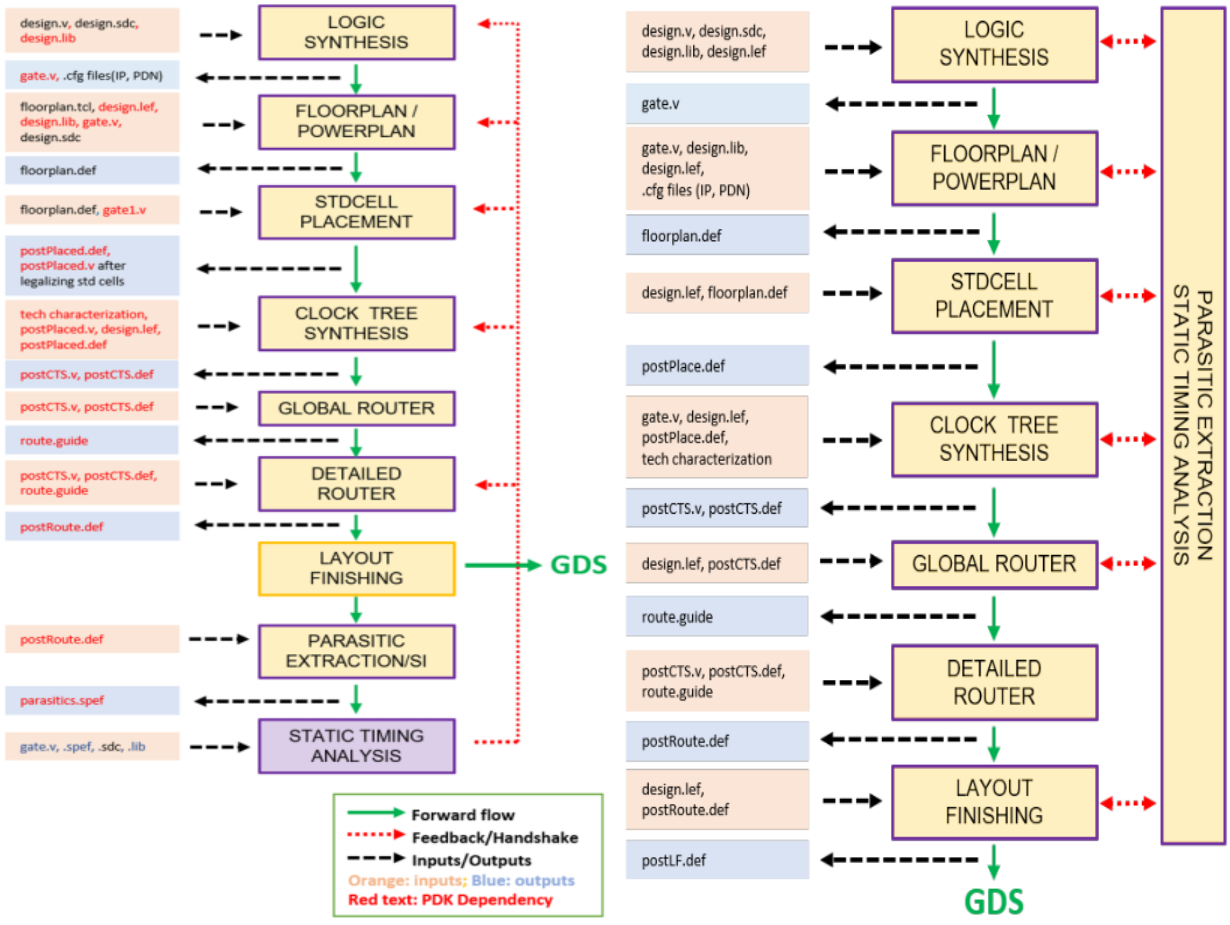
```
================================================================================
report_checks -path_delay min
--------------------------------------------------------------------------------
Startpoint: _134606_ (rising edge-triggered flip-flop clocked by core_clock)
```

```
Endpoint: _134606_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: min
# 此处略去具体路径
-------------------------------------------------------------
           0.00   data required time
          -0.13   data arrival time
-------------------------------------------------------------
           0.13   slack (MET)
=============================================================================
report_checks -path_delay max
-----------------------------------------------------------------------------
Startpoint: _129344_ (rising edge-triggered flip-flop clocked by core_clock)
Endpoint: _130077_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: max
# 此处略去具体路径
           5.56   data required time
          -2.04   data arrival time
-------------------------------------------------------------
           3.52   slack (MET)
=============================================================================
report_checks -unconstrained
-----------------------------------------------------------------------------
Startpoint: _129344_ (rising edge-triggered flip-flop clocked by core_clock)
Endpoint: _130077_ (rising edge-triggered flip-flop clocked by core_clock)
Path Group: core_clock
Path Type: max
# 此处略去具体路径
           5.56   data required time
          -2.04   data arrival time
-------------------------------------------------------------
           3.52   slack (MET)
report_tns
tns 0.00
report_wns
wns 0.00
=============================================================================
report_check_types -max_transition -all_violators
-----------------------------------------------------------------------------

=============================================================================
report_power
-----------------------------------------------------------------------------
Group                  Internal  Switching    Leakage      Total
```

```
                          Power       Power       Power       Power
-----------------------------------------------------------------------
Sequential              4.50e-03    4.50e-05    6.17e-04    5.16e-03  32.9%
Combinational           5.09e-03    3.31e-03    2.14e-03    1.05e-02  67.1%
Macro                   0.00e+00    0.00e+00    0.00e+00    0.00e+00   0.0%
Pad                     0.00e+00    0.00e+00    0.00e+00    0.00e+00   0.0%
-----------------------------------------------------------------------
Total                   9.59e-03    3.36e-03    2.75e-03    1.57e-02 100.0%
                           61.1%       21.4%       17.5%
=======================================================================
report_design_area
-----------------------------------------------------------------------
Design area 2822190 u^2 100% utilization.
```

Listing 11: final report



(a) (b)

图 6: The OpenROAD flow. (a)[ABC+19] 图 3, (b)[ACF+19] 图 1. `results` 目录中的各个结果文件和 flow 中各个步骤之间的对应关系。

4.3 `results` 目录与执行结果

该目录中存放着工具链运行产生的中间结果，和各个步骤的对应关系如图6所示。其中有五类文件：

1. 转化后的门级网表形式的 Verilog 代码`.v` 文件。
2. 各个步骤所需的时序约束`.sdc` 文件，其中的信息与 Listing 7：design.sdc 一致。**所以搞不清楚为什么生成这么多冗余的文件**。
3. 上文提到的知道布线细节的`.guide` 文件。
4. GDSII 版图文件，可以用 Klayout 工具打开观察版图, 如图7所示。使用的命令为如下：

```
$ klayout results/nangate45/rv32_toy/6_final.gds
```

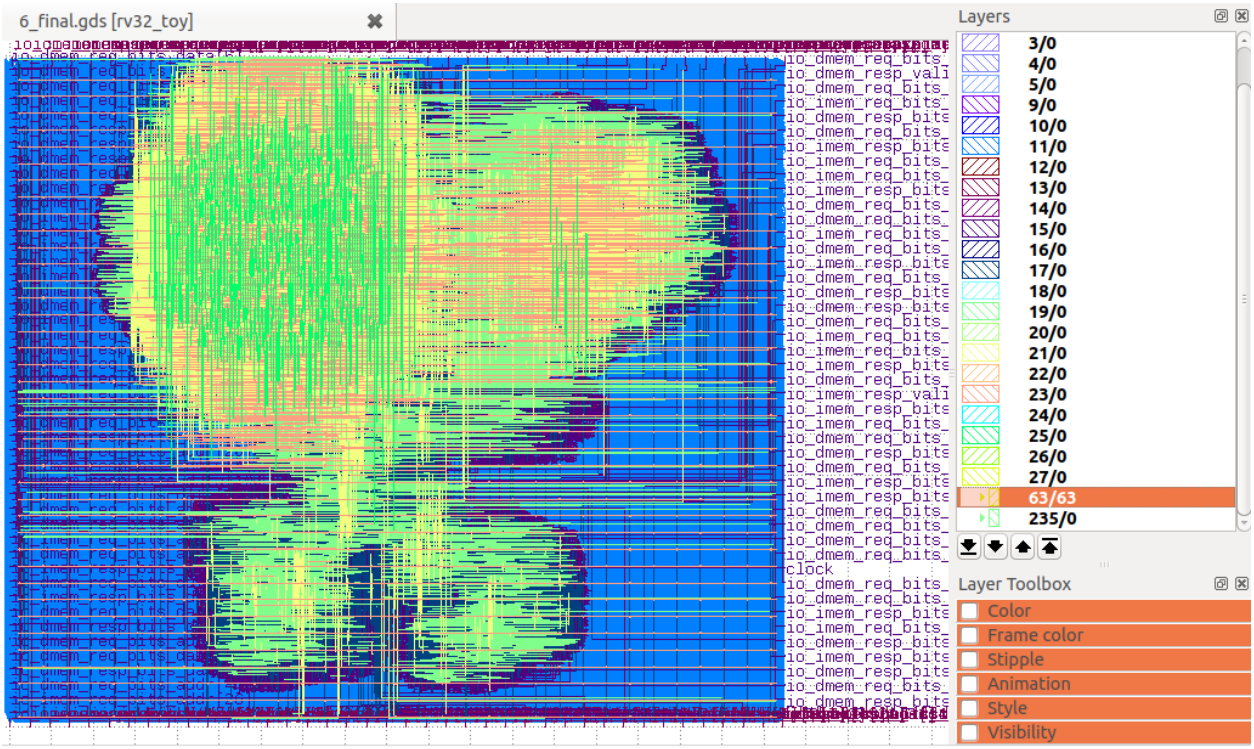


图 7: rv32_toy GDSII 版图，隐藏了最上面的两层。

5. `.def` 文件是工具链流程中产生的中间文件，用来作为工具链上游工具到下游工具信息的交互。这些`.def` 文件也是可以用 Klayout 打开的，打开后显示的就是工具链中间流程的版图半成品。这里我们以第五步 route 布线为例，用 Klayout 打开 `5_route.def` 文件的效果如图8所示。使用的命令如下：

```
$ klayout -nn objects/nangate45/rv32_toy/klayout.lyt results/nangate45/rv32_toy/5
    _route.def
```

```
$ # 或者用如下Makefile中已经提供的命令
$ make DESIGN_CONFIG=./designs/nangate45/rv32_toy.mk klayout_5_route.def
```

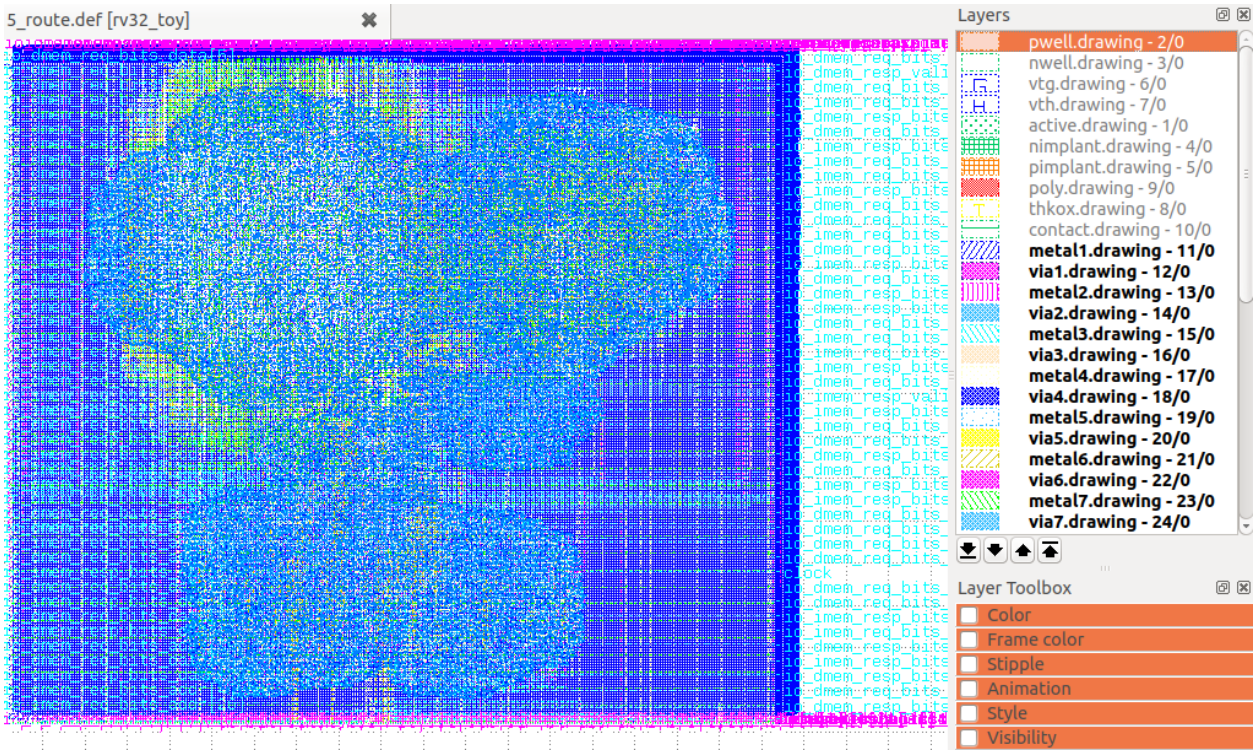


图 8: rv32_toy 第五步 route 布线的中间版图。

5 分析与问题

更多的处理器就代表了更多的可能性，也就有更大的可能性能够反应出这个开源工具链更多的问题。一旦解决了这些浮出的问题，EDA 开源工具链也就会变得越来越好。

5.1 分析

所以我一共选了 12 个 RTL 代码工程，其中 11 个是 CPU 的代码，还有一个 4KB 的 Icache 代码，如表2所示。表中名称标红的 7 个项目都是我自己用 Chisel 语言开发的，除了 Icache 外，均在章节3中有所介绍。而 Icache 设计为最简单的单路之间映射的组织形式，是命中一个周期即可返回指令的 L1 cache。Icache 的面积是可用参数设置的，而在试验中我也分别配置了 16KB 和 4KB 两个不同的容量来搭配 CHIWEN 处理器（另外可以搭配 FUXI 和 BIAN，因为我的 Icache 还可以配置双读端口还是单读端口；但是这里我们只是观察 EDA 后端运行的效果，搭配在哪个处理器上个人感觉影响不是那么大）。单独测试的 Icache 只有 4KB 版本的，没有 16KB 的。这个在问题部分再来讨论。剩下的 5 个项目是公布在 Github 上的开源项目，分别是：tinyRocket（OpenRoad 工具链自带的示例 project），swerv（OpenRoad 工具链自带的示例 project），picorv32, ultraembedded,

表 2: 各个 project 的关键配置和报告信息。其中时序栏中上下分别为设置的周期和最大延时（正常情况下参考 `6_final_report.rpt` 文件给出的路径时延数据；但是带 * 号表示整个流程并未完成，所以参照的是文件 `4_1_cts_prefillcell.log` 文件给出的路径时延数据；目前已知的情况为 `4_1_cts_prefillcell.log` 和 `6_final_report.rpt`【如果有】给出的时序报告是一致的）；设置面积栏中上下分别为 die area 和 core area 的具体设置（`.mk` 文件中）；资源开销栏中上下分别是 cell（又称 component）的数量和 net（又称 wire bit）的数量（参考 log 文件 `5_1_fastroute.log`，数据和流程后续的 log 文件一致）；实际面积栏上下分别是电路真实面积和其占设置面积的比例，也即利用率（参考 log 文件 `4_1_cts_prefillcell.log` 最后给出的 Design Stats 统计数据）；最后四栏为功耗（参考 `6_final_report.rpt` 文件中的功耗分析；??? 表示因为工具链未能执行完成而未知）。

名称	时序 设置面积 die area core area	资源开销 cells nets	实际面积 movable area util/padded	内部功耗	翻转功耗	漏电功耗	总功耗
picorv32	5.6 2.84 (0 0 320.15 320.6) (10.07 11.2 310.27 310.8)	57423 16321	24458.7 27%–49%	7.81e-04	6.26e-04	5.45e-04	1.95e-03
ultraembedded	5.6 2.33 (0 0 924.92 799.4) (10.07 9.8 914.85 789.6)	205275 39434	59835.9 8%–15%	1.42e-03	1.25e-03	1.42e-03	4.09e-03
tinyRocket	5.6 2.80 (0 0 924.92 799.4) (10.07 9.8 914.85 789.6)	216332 44116	65874.4 9%–17%	5.62e+01	3.21e-03	1.84e-03	5.62e+01
rv32_toy*	5.6 2.04 (0 0 924.92 799.4) (10.07 9.8 914.85 789.6)	309899 79069	121698.5 17%–31%	9.59e-03	3.36e-03	2.75e-03	1.57e-02
rv32_ttoy	10.0 2.09* (0 0 1550.02 1342.6) (10.07 11.2 1540.14 1332.8)	532799 93682	143333.6 7%–13%	???	???	???	???
FUXI	10.0 2.13 (0 0 1550.02 1342.6) (10.07 11.2 1540.14 1332.8)	624208 126092	188566.9 9%–17%	1.08e-02	2.96e-03	4.39e-03	1.81e-02
swerv	10.0 3.07 (0 0 1550.02 1342.6) (10.07 11.2 1540.14 1332.8)	679501 147491	232049.1 11%–20%	2.63e-03	2.52e-03	5.18e-03	1.03e-02
BIAN	10.0 2.29* (0 0 2200.01 2199.4) (10.07 11.2 2189.94 2189.6)	1399730 273538	418707.7 9%–16%	???	???	???	???
ridecore	30.0 3.24* (0 0 2200.01 2199.4) (10.07 11.2 2189.94 2189.6)	2060259 532157	936080.3 20%–33%	???	???	???	???
Icache4KB	10.0 2.88 (0 0 1550.02 1342.6) (10.07 11.2 1540.14 1332.8)	904330 241667	431028.0 21%–35%	7.72e-03	7.39e-03	1.02e-02	2.53e-02
CHIWEN4KB	10.0 3.20 (0 0 2200.01 2199.4) (10.07 11.2 2189.94 2189.6)	1562265 354967	595970.1 13%–21%	1.36e-02	9.53e-03	1.40e-02	3.71e-02
CHIWEN16KB	10.0 10.02* (0 0 3100.04 2683.2) (10.07 11.2 3080.28 2665.6)	4210698 1190668	2198249.3 27%–44%	???	???	???	???

ridecore。tinyRocket 是经典的五级流水线架构，和我自行设计的 rv32_toy、CHIWEN 不带 Icache 对标；swerv 是 32bit 双发射 9 级流水线顺序执行架构，和我自行设计的 rv32_ttoy、FUXI 对标；ridecore 是乱序双发射架构，和我自行设计的 BIAN 对标。剩下的两个处理器：picorv32 在 project 说明中有提到，所以我就用 OpenRoad 运行也运行了一遍；ultraembedded 是一位同学在群里问的，抽了个空也运行了一遍。picorv32 是一款由著名的 IC 设计师 Clifford Wolf（Yosys 工具的创始开发者）开发并开源的一款 RISC-V 处理器核；其追求低面积与高主频，所以是多周期的架构。ultraembedded 则是五级流水，但是在译码级新增了一个 stall buffer；在执行接新增了一个 scoreboard 计分板。另外，这几个开源的处理器核除了 tinyRocket 都不是自带 L1 cache 的。将这 12 个 project 的一些关键设置和关键的衡量指标（最大延迟、资源使用量、面积、功耗）整理为表2。

表2中的 12 个 project 从上到下占用的资源和实际的面积逐渐增长，代表着结构的越来越复杂（最后带 Icache 的 3 个自研 project 除外）；可以按照结构大致划分为五类：

1. picorv32 单独成一类，为多周期结构。特点是资源的开销、实际面积和功耗都极小，大幅度的小于其他 project。但是实测的时序，也即最大时延却没有预期的那么好，能够达到的最高主频在 12 个 project 中处于偏下的水准。

2. ultraembedded, tingRocket 和 rv32_toy 成一类，为单发射顺序流水线结构。虽然我自行设计的处理器 rv32_toy 的时序最好。但是其他方面——资源的使用量、面积和功耗都是 3 款处理器中最高的（tingRocket 的功耗显然是将 Icache 和 Dcache 也算入在内了，所以比其他处理器都高出了几个数量级）。这一点非常糟糕。但是奇怪的是，rv32_toy 明明是我所能设计的结构最为简单的五级流水处理器了，转移预测机制简单，没有乘除法器。但是使用的资源和占用的面积却明显比其他两个处理器来得高。所以一定要精确定位到问题出在了哪里。

3. rv32_ttoy, FUXI 和 swerv 成一类，为双发射顺序流水线结构。rv32_ttoy 其实和 FUXI 结构基本一致，但是 FrontEnd 中的转移猜测和取值的逻辑都做了简化，这一点可以从资源和面积的占用上就能看的出来。时序上，我自行设计的两款处理器 rv32_ttoy 和 FUXI 要比 swerv 好很多；而且 swerv 因为其是 9 级流水，结构比我设计的要复杂很多，占用的资源和面积也自然比我的要高。但是同样的问题——我的设计不知道莫名地功耗大，比 swerv 的功耗高出近一倍（问题出在了内部功耗上，但是深层的原因还尚不可知）。

4. BIAN 和 ridecore 成一类，为双发射乱序结构。无论是从时序上，还是从资源面积上，我设计的 BIAN 都要比开源的 ridecore 要出色。但是遗憾的是，两款处理器都无法执行完成整个流程。

5. 最后是特殊的一类，由 Icache, CHIWEN4KB 和 CHIWEN16KB 组成。目的在于研究由大量 reg 堆叠的简单存储对于 OpenRoad 工具链编译 RTL 设计的影响。不难发现，各个方面的影响都是显著的。OpenRoad 不会自动识别这些存储单元，替换成库中的 sram 组件。

通过 OpenRoad 开源的 EDA 工具链，还有一些零星的观察：

- 最让我惊喜的当属 BIAN 的时序了，没想到在 ASIC 上能够这么好，着实吃惊。在 BIAN 设计中有很多耗时的旁路，所以其实我个人之前一直对 BIAN 是不太满意的，有空肯定会重写一版！
- 但是我的设计一个很大的缺陷在于功耗都比同样类型的其他开源设计要大不少。这是必须定位到问题根源并解决的。
- 和 rv32_toy 相比，rv32_ttoy 最大的区别在于 BackEnd 有两套流水线，其他的改动包括 FrontEnd 的改动都不大。但是这两者的资源和面积使用量相差并不大，尤其是面积。说明流水线本身是非常经济的。
- 关于 OpenRoad 的运行时间，我挑了几个 project 做了大致的统计，其中：picorv32 仅用时 3 分钟，tinyRocket 用时 10 分钟，swerv 用时 45 分钟，Icache 用时 94 分钟。所以其用时和实际的面积基本上成正比。
- OpenRoad 不会到用 Reg 堆叠的 Memory 有任何的优化。如果需要带上 L1 cache 运行 OpenRoad，推荐的方式是像 tinyRocket 一样，使用 OpenRoad 提供的库模块 `fakeram45`。
- tinyRocket 正是用库模块 `fakeram45` 来搭建 Icache 和 Dcache 的。OpenRoad 在资源和面积的报告中是不包括这些 Memory 的，甚至在最后的 GDSII 版图上都看不出来（见图17，所以处理器核才会被挤到左上角的角落处）；时序的报告是否包括 cache 访存尚不可知；功耗的报告一定是包括 cache 的，因为和其他处理器相比，功耗高出了几个数量级，且绝大部分来自内部功耗。
- 面积越大，结构越复杂，占用的内存就越大，生成的 GDSII 文件也越大。运行 FUXI 使用内存的峰值是 1.2GB；而运行 CHIWEN4KB 使用的内存峰值是 5GB。FUXI 的 GDSII 文件大小为 131.9MB；CHIWEN4KB 的 GDSII 文件大小为 360MB；最小的 picorv32 仅为 13.2MB。
- 从报告中也可以看到一些处理器设计的特殊之处。比如 swerv 中使用到了异步逻辑；而 ridecore 中使用了时钟下降沿，所以将时钟二分频了，效果如下：

```
        15.00   15.00   clock core_clock' (rise edge)
         0.00   15.00   clock network delay (ideal)
         0.00   15.00   clock reconvergence pessimism
        15.00 ^ _944025_/CK (DFF_X1)
-0.04   14.96   library setup time
        14.96   data required time
-----------------------------------------------------------
        14.96   data required time
        -3.24   data arrival time
-----------------------------------------------------------
        11.72   slack (MET)
```

Listing 12: `4_1_cts_prefillcell.log`

表2显示了我们为 ridecore 设置的时钟周期为 30ns, 但这里只有 15ns。此外 ridecore 的时序分析在不同步骤里面出现了非常严重的分歧。上面 Listing 12 展示的是 `4_1_cts_prefillcell.log` 流程步骤 4.1 的分析数据；下面 Listing 13 展示的是 `3_post_resize.rpt` 流程 3.2resize 之后的分析数据。最长路径的时延相差了 10ns 有余。

```
        15.00   15.00   clock core_clock' (rise edge)
         0.00   15.00   clock network delay (ideal)
         0.00   15.00   clock reconvergence pessimism
        15.00 ^ _926072_/CK (DFF_X1)
-0.04   14.96   library setup time
        14.96   data required time
-----------------------------------------------------------
        14.96   data required time
       -13.56   data arrival time
-----------------------------------------------------------
         1.40   slack (MET)
```

Listing 13: `3_post_resize.rpt`

5.2 问题

通过上述的 12 个具体 project,在使用 OpenRoad 工具链的过程中,遇到的问题是较多的,表2中就有 4 个工程执行失败，而且均失败在了步骤 5.1 Run global route（在 log 中被称为 FastRoute),用到的工具是 UTD-BoxRouter。所以 FastRoute 成为了不折不扣的“鬼门关”。具体的错误形式又可以细分为 3 种:

1. 在 rv32_ttoy 和 BIAN 中表现出来的，日志文件 `5_1_fastroute.log` 中的记录如下:

```
> Running FastRoute...
> --Running extra iterations to remove overflow...
# 此处略去很多迭代步
> ----updateType 4
> ----iteration 100, enlarge 369, costheight 679, threshold 0 via cost 0
>---log_coef 0.838120, healingTrigger 33 cost_step 2 L 1 cost_type 1 updatetype 4
[ERROR] FastRoute cannot handle very congested design
Command exited with non-zero status 2
```

Listing 14: rv32_ttoy

```
> Running FastRoute...
> --Running extra iterations to remove overflow...
# 此处略去很多迭代步
> ----updateType 4
> ----iteration 100, enlarge 523, costheight 689, threshold 0 via cost 0
>---log_coef 0.953011, healingTrigger 33 cost_step 2 L 1 cost_type 1 updatetype 4
[ERROR] FastRoute cannot handle very congested design
```

```
Command exited with non-zero status 2
```

Listing 15: BIAN

都是迭代到 100 轮上界之后程序被强行终止了。具体问题出在哪里尚不可知，但这绝对是 OpenRoad 工具链的一个 Bug。因为从表2中来看，处理器 rv32_ttoy 真的就没有 FUXI 那么复杂，谈何 very congested design 之说呢？凭什么 FUXI 能够运行通过得到最后的 GDSII 文件和 final report 而 rv32_ttoy 却得不到呢？报出的 ERROR`FastRoute cannot handle very congested design` 显然只是一个表面现象。而且我还观察到一些有意思的现象。这两个工程在这一步都会先卡在 16 轮迭代后；通过 16 轮之后，接着又会卡在 39 轮；接着卡在 62；接着卡在 85；最后到 100 轮被终止。可以发现一个 23 轮迭代的周期。另外 16 轮后如果卡了，基本就凉了；OpenRoad 基本就不会给你通过的机会。跑完整个 100 轮迭代最后程序终止需要很长的时间；在我的笔记本电脑上都是运行一天左右的。此外之前 rv32_toy 也报出过这样的错误，但可惜的是我只存档了报错的日志，如下：

```
> Running FastRoute...
> --Running extra iterations to remove overflow...
# 此处略去很多迭代步
> ----updateType 4
> ----iteration 100, enlarge 369, costheight 855, threshold 0 via cost 0
> ----log_coef 2.000000, healingTrigger 16 cost_step 5 L 1 cost_type 1 updatetype
    4
[ERROR] FastRoute cannot handle very congested design
Command exited with non-zero status 2
```

Listing 16: rv32_toy

后来经过一顿改之后，终于能够成功将 OpenRoad 流程执行下来了，但是那个引发错误的代码也被覆盖掉了（但其实我该的地方非常少）。后来再也复现不出来了，就此失去了一个准确定位到代码逻辑的机会。感觉自己亏了一个亿！

2. 在 ridecore 中表现出来的，日志文件 `5_1_fastroute.log` 中的记录如下：

```
Notice 0:     Created 259 pins.
Notice 0:     Created 2060259 components and 5691354 component-terminals.
Notice 0:     Created 5 special nets and 4120518 connections.
Notice 0:     Created 532157 nets and 1570836 connections.
Notice 0: Finished DEF file: ./results/nangate45/pipeline/4_cts.def
Adjust layer 2 in 50.0%
Adjust layer 3 in 50.0%
Command terminated by signal 11
```

Listing 17: ridecore

这个错误莫名奇妙的，什么信息都没有反应出来。

3. 在 CHIWEN16KB 中表现出来的，日志文件 `5_1_fastroute.log` 中的记录如下：

```
Notice 0: Finished DEF file: ./results/nangate45/pipeline/4_cts.def
Adjust layer 2 in 50.0%
Adjust layer 3 in 50.0%
# 位置和上述的ridecore一致，但是这里不会终止，而是会一种卡住，执行不下去
```

Listing 18: CHIWEN16KB

个人认为 OpenRoad 整个 flow 最薄弱的一个环节就是这个 5.1 FastRoute。对应工具的代码是要重点开发升级的。群里面也有很多同学提出：通过不断修改面积最后执行成功了，但个人感觉这种方法治标不治本，往往会忽略了工具链中潜在的 BUG。更致命的是，我连`.mk` 文件中需要设置的面积的参数具体含义都不明确，只是按照 OpenRoad 已经提供的连`.mk` 文件依葫芦画瓢而已。在这种艰难的情况下不断的调试不同的参数无异于是盲人摸象。无论是我自行设计的 BIAN 或者是开源的 ridecore，像这种结构已经变得复杂的处理器，在 OpenRoad 上都“阵亡”了，不得不让人质疑 OpenRoad 对大型复杂 RTL 工程尤其是后端物理设计上支持的能力。

最后我再提几个零星的问题：

- 在 `flow/designs/src/tiny-tests` 目录下的无论运行 LargePinCount 还是 SmallPinCount 都存在问题。LargePinCount 的问题在于：

```
ERROR: number of pins (10002) exceed max possible (3976)
Command exited with non-zero status 1
```

不过一般设计中也没有那么多的引脚，不然也不好封装。SmallPinCount 暴露出来的问题就要棘手很多：

```
clock
[WARNING] Net "clock" has 1 sinks. Skipping...
[ERROR] No valid clock nets in the design. Exiting...
```

可是如果找到 SmallPinCount 的源码 `SmallPinCount.v`，如 Listing 19 所示，明明就是有时钟 clock 信号的，为什么会被规约掉？

```
module SmallPinCount(input clock,
    input [5:0] io_wide_bus,
    output reduced);
  reg reduce_it;
  assign reduced = reduce_it;
  always @(posedge clock) begin
    reduce_it <= |io_wide_bus;
  end
endmodule
```

Listing 19: `SmallPinCount.v`

另外群里有同学反映 OpenRoad 运行纯组合逻辑会报错也是一个问题。

- 在 OpenRoad 的报告中，虽然会有时序和功耗的数据，但是我又怎么能定位到源码进行重新设计与调优呢? 在报告中的最长路径怎么能和原 RTL 代码对应呢? 能不能有像 Xilinx Vivado 工具一样有将最长路劲可视化出来的功能。这将会给芯片的迭代调优带来极大的便利。
- 最后，不得不承认，自己对设置的参数，每个步骤的 tcl 脚本的具体执行任务，每个工具的使用方法，以及执行过程中产生的信息是严重的了解不足。

致谢

感谢朱威浦同学为 `demo` 工程搭建了一个简易的行为仿真平台。感谢王重熙同学在仿真调试的过程中指出了 rv32_toy 设计中的一个寄存器未初始化导致的 Bug。另外，`demo` 工程源于另外一门 EDA 课程的项目设计，朱威浦和王重熙与我共属于一个小组，所以会有这种合作。

展望

近的来说，当务之急就是搞清楚在步骤 5.1 FastRoute 的 Bug 出在哪里。更长远地来看，希望能够为国产的 EDA 工具贡献出一份力量，不用再受制于人。谁不想命运掌握在自己的手里呢?

A 版图

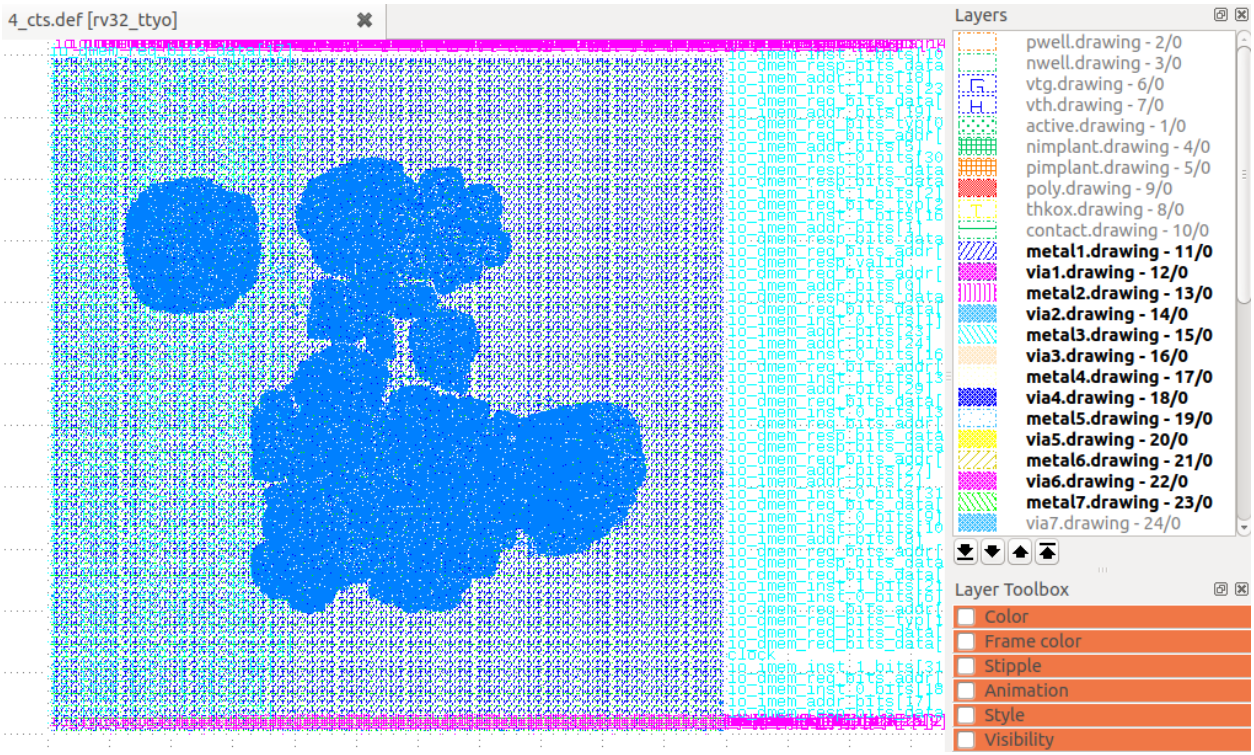


图 9: BIAN 第四步 CTS 时钟树综合的中间版图

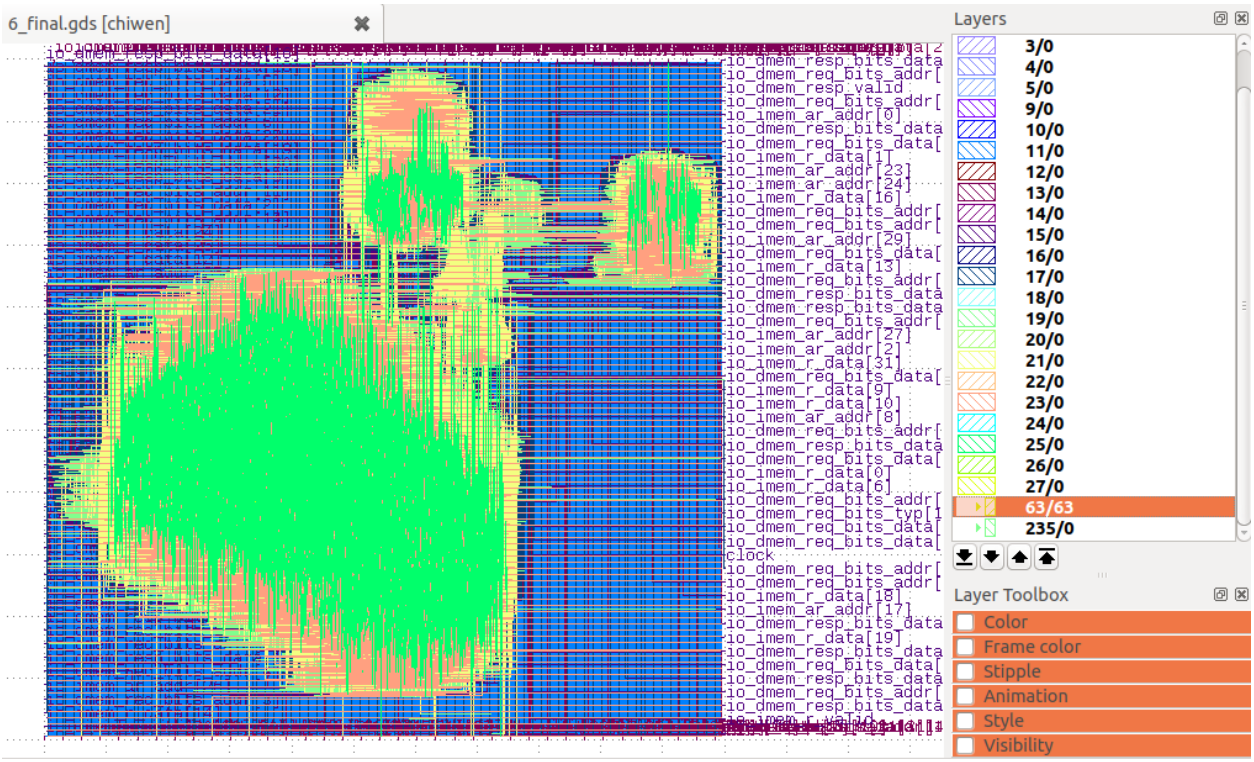


图 10: CHIWEN GDSII 版图，隐藏了最上面的两层。

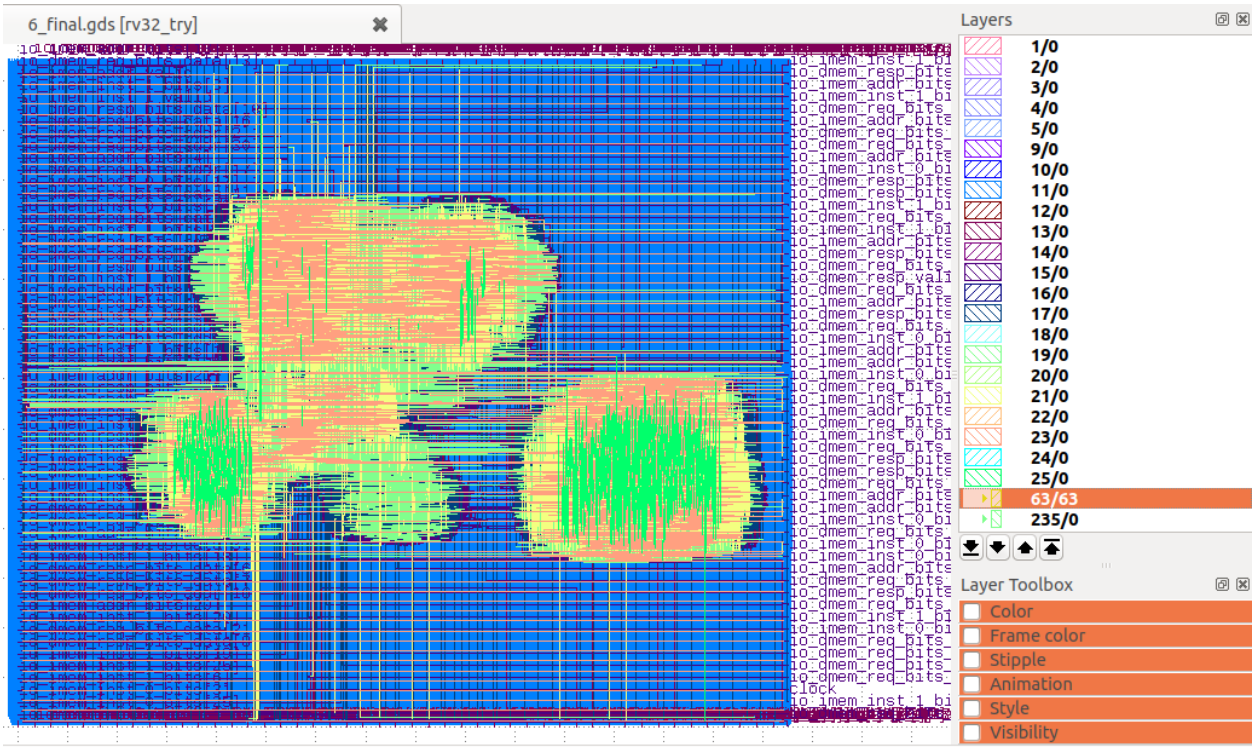


图 11: FUXI GDSII 版图，隐藏了最上面的两层。

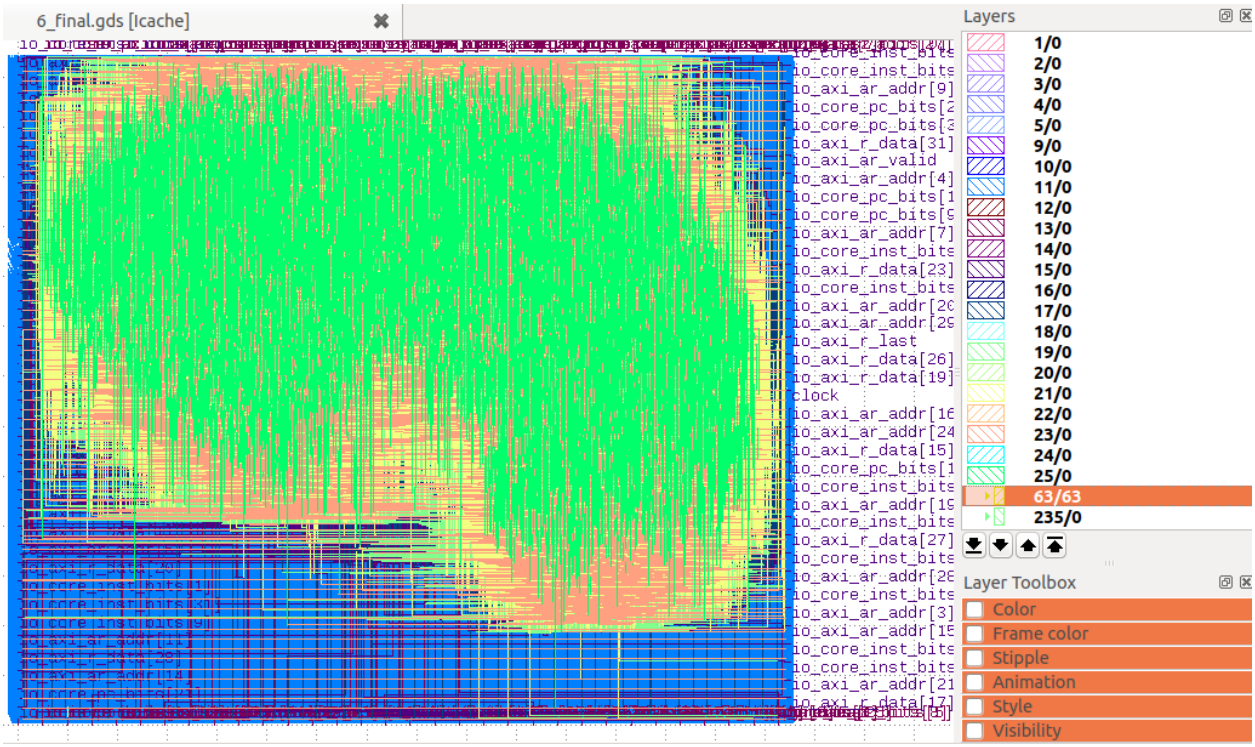


图 12: Icache GDSII 版图，隐藏了最上面的两层。

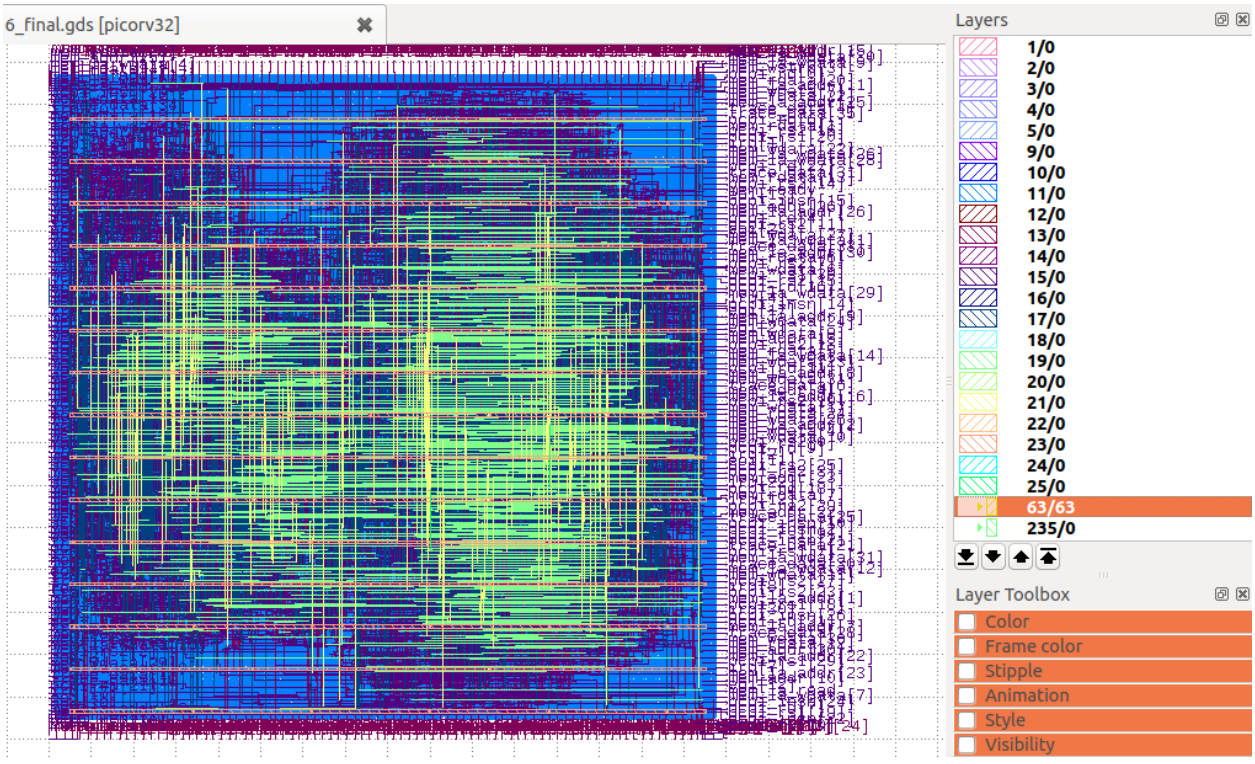


图 13: picorv32 GDSII 版图，隐藏了最上面的两层。

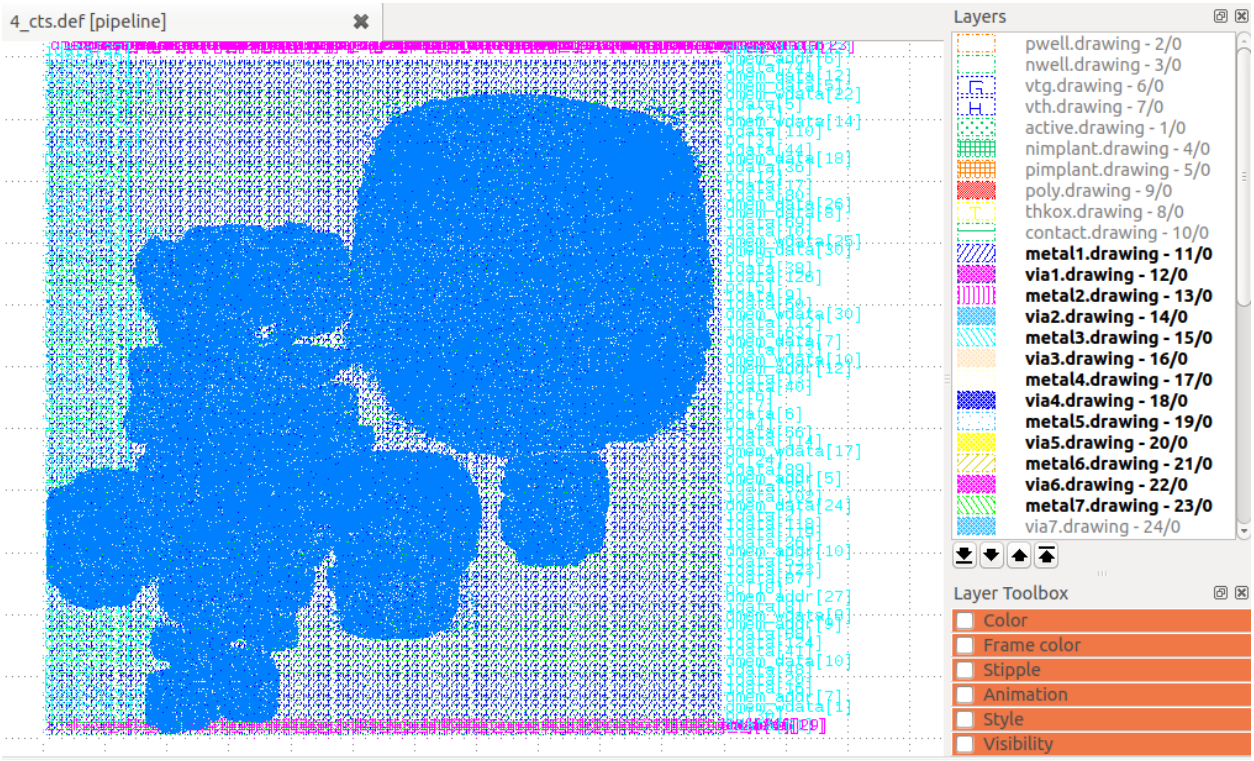


图 14: rv32_toy 第四步 CTS 时钟树综合的中间版图。

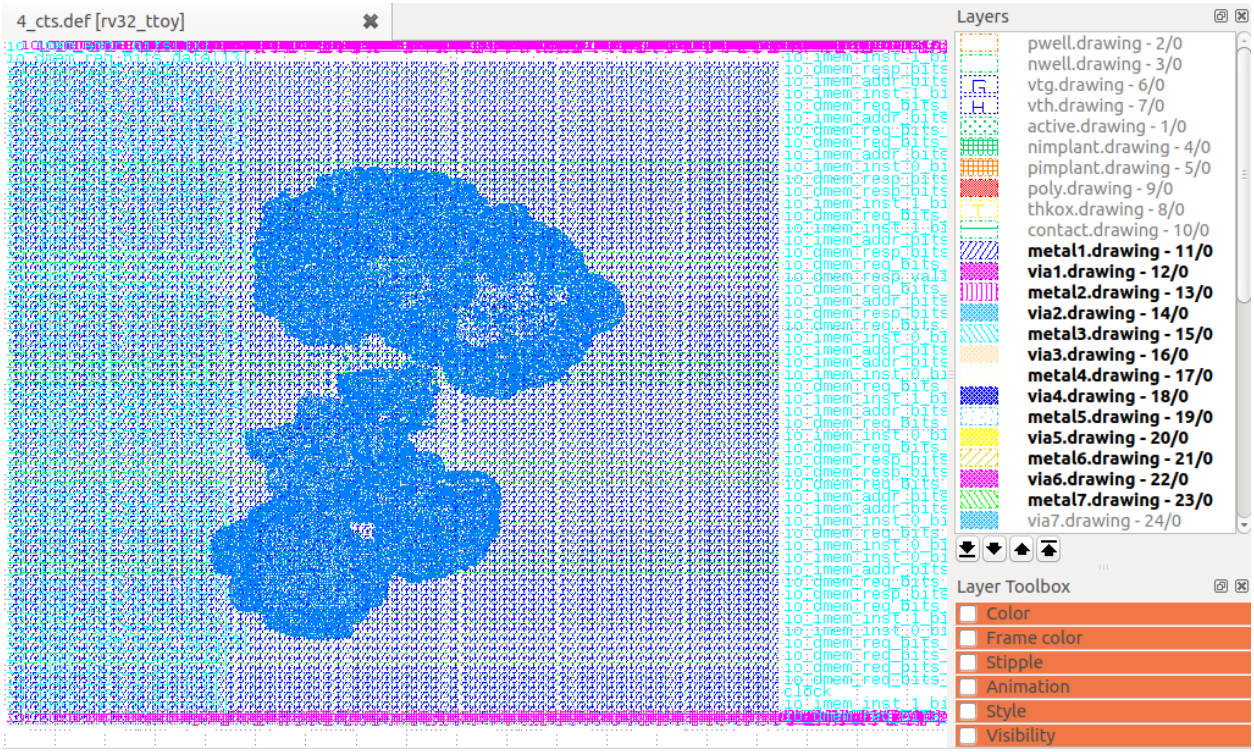


图 15: rv32_ttoy 第四步 CTS 时钟树综合的中间版图

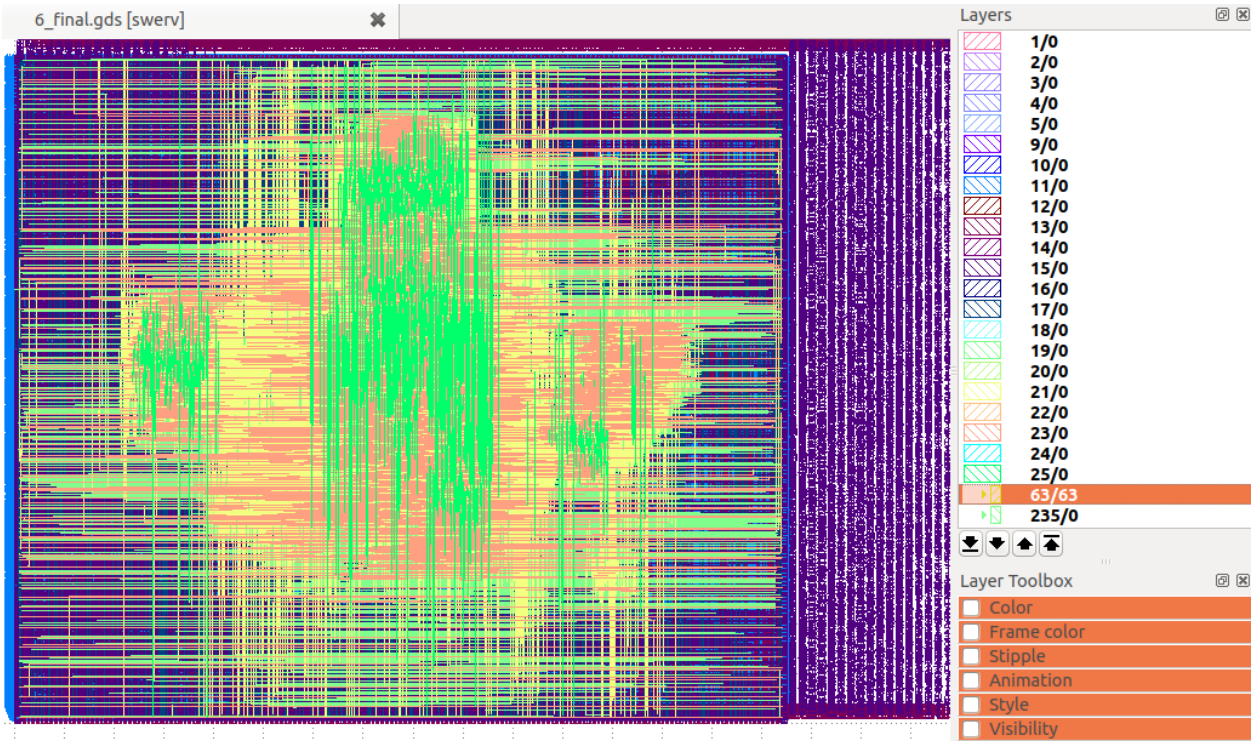


图 16: swerv GDSII 版图，隐藏了最上面的两层。

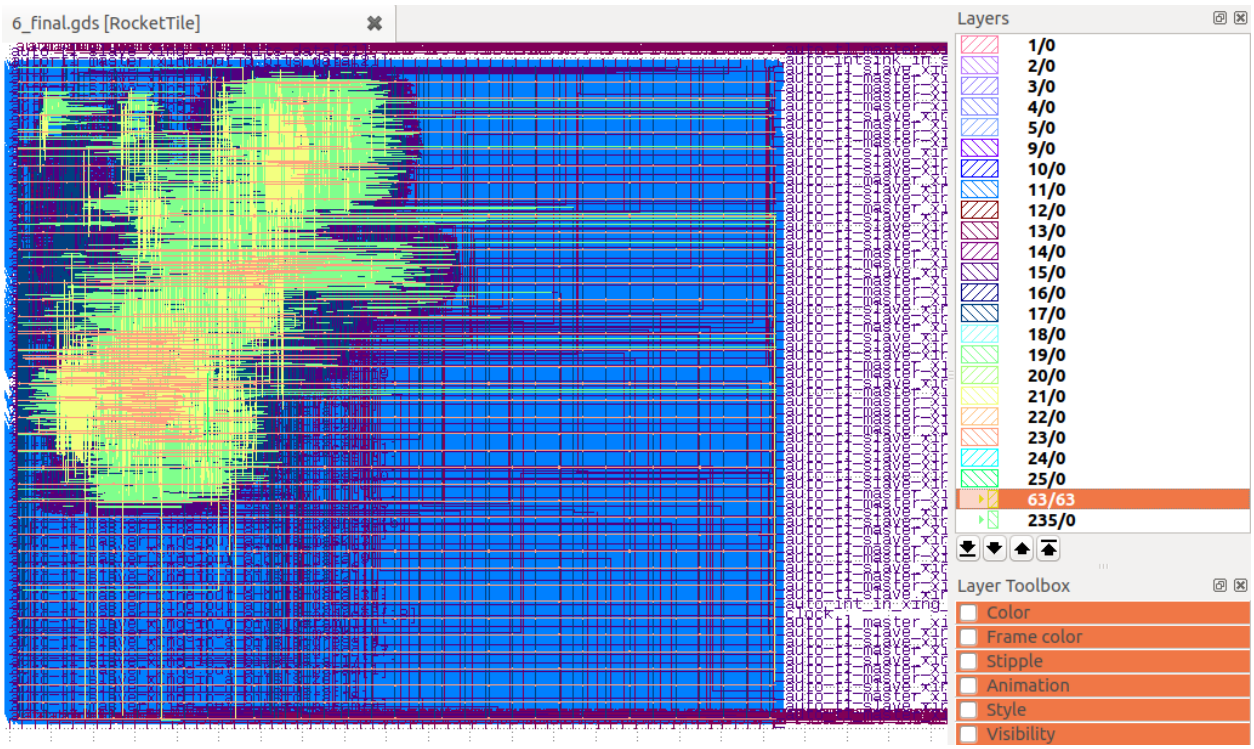


图 17: tinyRocket GDSII 版图，隐藏了最上面的两层。

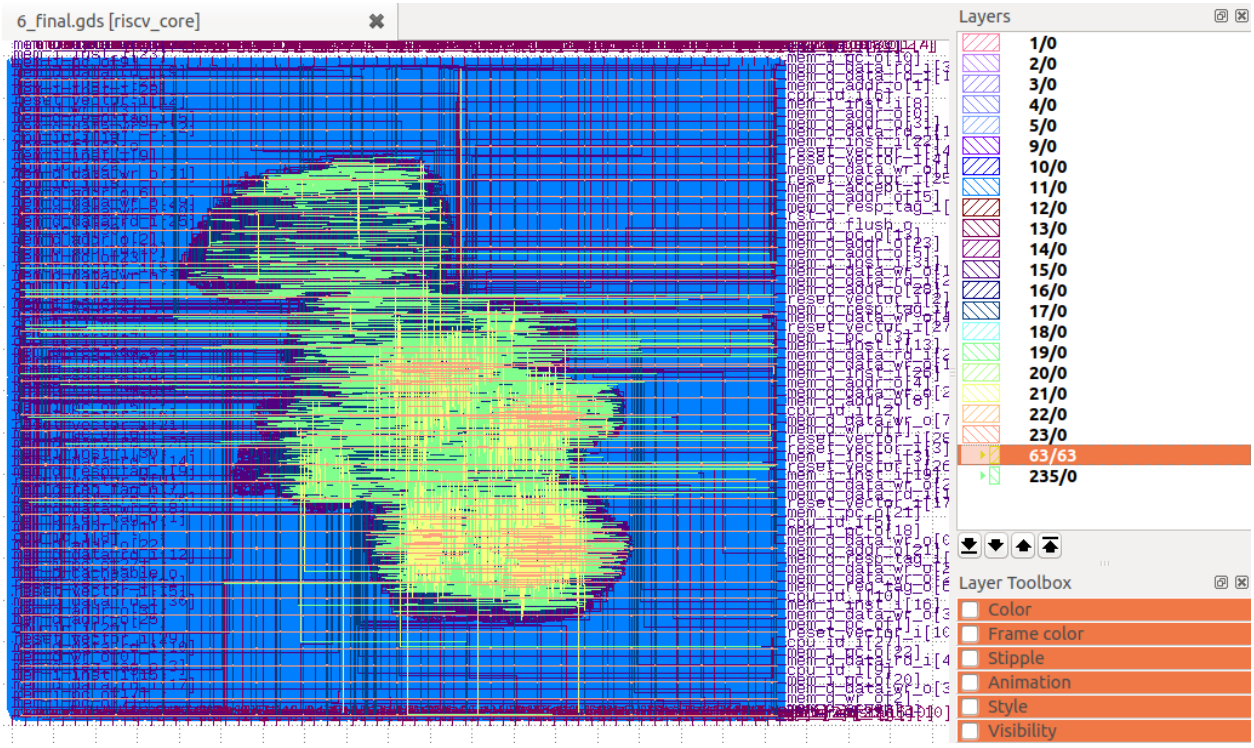


图 18: ultraembedded GDSII 版图，隐藏了最上面的两层。

参考文献

[ABC+19] T Ajayi, D Blaauw, TB Chan, CK Cheng, VA Chhabria, DK Choo, M Coltella, S Dobre, R Dreslinski, M Fogaça, et al. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. *Proc. GOMACTECH*, pages 1105–1110, 2019.

[ACF+19] Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–4, 2019.

[AM03] Saurabh N Adya and Igor L Markov. Fixed-outline floorplanning: Enabling hierarchical design. *IEEE transactions on very large scale integration (VLSI) systems*, 11(6):1120–1135, 2003.

[CKKW18] Chung-Kuan Cheng, Andrew B Kahng, Ilgweon Kang, and Lutong Wang. Replace: Advancing solution quality and routability validation in global placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(9):1717–1730, 2018.

[HKL18] Kwangsoo Han, Andrew B Kahng, and Jiajia Li. Optimal generalized h-tree topology and buffering for high-performance and low-power clock distribution. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[KWX18] Andrew B Kahng, Lutong Wang, and Bangqi Xu. Tritonroute: an initial detailed router for advanced vlsi technologies. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[周 19] 周盈坤. 乱序双发射处理器的自主设计实现. 中国科学院大学本科毕业论文, 2019.