计算机系统综合实验手册

目录

1 基本工具链

本节所涉及的内容，是为开展计算机系统综合实验的必要基础，请务必完全熟练掌握。

1.1 git 基本用法

git 的教程有很多，这里不详细叙述 git 的各种基本运作机制，仅从样例出发，讲解必要掌握的命令。

首先是最基本的 clone 项目：

```
git clone https://github.com/your/repo
git clone git@github.com:your/repo # 需将你的公钥上传到github上
```

上传公钥：

```
# 先生成密钥
~bash> cd ~/.ssh && ls -al
. ..
~bash> ssh-keygen -t rsa -C "you@email.cn"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/you/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/you/.ssh/id_rsa.
Your public key has been saved in /home/you/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:ORKaY5n3N4G0P4guAbT0UFNZNorA3mqSL0S0xPWo/+U you@email.cn
The key's randomart image is:
+---[RSA 2048]----+
|..ooo..o+        |
| +=.oo.o .       |
|o+.*..o .        |
| o= o= o +       |
|.o oB o S .      |
|o.+..o + + .     |
|.+ . ...o =      |
|. . o.o  . o     |
| .   o.E         |
+----[SHA256]-----+
~bash> ls # 现在可以看到id_rsa.pub文件
id_rsa  id_rsa.pub
~bash>
```

进入 github 的 settings -> SSH and GPG keys -> New SSH Key，然后将 id_rsa.pub 的内容拷贝进去即可。

除了克隆项目，还有如下必要用法：

```
git push # 推送
git pull # 拉取
```

对远程仓库的管理：

```
git remote -v # 查看远程仓库的情况

# 绑定一个远程仓库git@github.com:new/repo并将其命名为origin
git remote add origin git@github.com:old/repo

# 将仓库名origin所绑定的远程仓库地址更改为git @github.com:new/repo
git remote set-url origin git @github.com:new/repo

# 将本地的local分支推送到远程的remote分支
git push origin local:remote

# 将本地的local分支强制推送到远程的remote分支，如有冲突，覆盖掉远程的项目
git push -f origin local:remote

# 加上-u可以设置缺省推送参数，即后续只需git push即可
git push -u origin local:remote
```

对本地项目的管理：

```
# 保存当前目录所有更改并提交到本地仓库
git add . -A && git commit -m 'message'

# 追加提交
git commit --amend -m 'new message'

# 用352cdeca2caa8a309d2所保存的文件覆盖掉当前目录
git checkout 352cdeca2caa8a309d2 .

# 查看352cdeca2caa8a309d2这一提交时README.md的内容
git show 352cdeca2caa8a309d2:./README.md
git show 352cdeca2caa8a309d2:./README.md | vim -

git log # 查看本地项目的变更历史
git reflog # 查看本地项目的完整变更历史，包括所有reset都记录在内
git reset --hard 352cdeca2caa8a309d2 # 回退到352cdeca2caa8a309d2，同时回退目录下的
    文件
git reset --soft 352cdeca2caa8a309d2 # 只回退git记录，不回退文件内容
```

1.2 Makefile

写在前面，对于过于复杂的 Makefile，可以使用`make -nB target`来查看生成`target`时会执行到的指令，这对于了解一个 Makefile 究竟干了什么非常有用。除此以外 Makefile 里面可

以通过$(info "xxx")来打印字符串，这个对于调试 Makefile 非常有用。

1.2.1 规则依赖

一个简单的例子：

```
a.o: a.c
  g++ a.c -o a.o

b.o: b.c
  g++ b.c -o b.o

binary: a.o b.o
  echo + binary
```

将上述内容写入到你当前目录的 Makefile 文件中，并保证a.c和b.c文件存在，然后在命令行里敲make binary，这条语句的意思是制造binary这个目标，make 会读特定名称的文件来获取制造这个目标的规则 (Makefile 是 make 内置的一个特定名称)。make 读取你的 Makefile 后会发现 binary 依赖于a.o和b.o，并进而计算得出a.o依赖于a.c，b.o依赖于b.c，最后按照你给定的生成规则g++ a.c -o a.o和g++ b.c -o b.o去生成相应的a.o和b.o。

敲完make binary之后，你 binary 并没有生成，因为你还没有为 binary 添加生成规则，所以如果你再敲一遍make binary，相应的echo + binary语句会被再执行一遍。

我们稍微改一下，让他变成下面这样：

```
a.o: a.c
  g++ a.c -o a.o

b.o: b.c
  g++ b.c -o b.o

binary: a.o b.o
  cat a.o b.o > binary
```

这个时候如果你在敲make binary，生成规则会将a.o与b.o拼接成 binary 文件，由于 binary 文件是最新生成的，它的时间戳大于a.o和b.o的时间戳，所以再下一次你敲make binary的时候，make 会输出'binary' is up to date。

一般而言，我们并不希望我们所生成的中间文件和源码文件共用一个目录，假设我们希望将目录结构调整成这样：

```
+---src
|   +--- a.c
|   +--- b.c
|
+---build
    +--- a.o
    +--- b.o
```

```
    +--- binary
```

为了达成这个目的，我们需要对 Makefile 做一些微调：

```
build/a.o: src/a.c
  mkdir -p build
  g++ src/a.c -o build/a.o

build/b.o: b.c
  mkdir -p build
  g++ src/b.c -o build/b.o

build/binary: build/a.o build/b.o
  cat build/a.o build/b.o > build/binary
```

相应的，基于这个 Makefile 来生成 binary，我们需要执行`make build/binary`，这条命令会按照你的规则去生成`build/a.o`和`build/b.o`两个文件，然后拼接成`build/binary`这个文件。

这个时候，你可能发现，make 的时候连目录一起写进去太撒刁了，所以这个 Makefile 可以进一步改进：

```
.PHONY: binary

binary: build/binary

build/a.o: src/a.c
  mkdir -p build
  g++ src/a.c -o build/a.o

build/b.o: b.c
  mkdir -p build
  g++ src/b.c -o build/b.o

build/binary: build/a.o build/b.o
  cat build/a.o build/b.o > build/binary
```

`.PHONY`的作用是定义一个伪目标，什么是伪目标，直观的讲就是，这个目标在磁盘上不存在。之前 Makefile 里面的`a.o`、`build/binary`、`build/a.o`这些目标在磁盘上是会对应到一个实际存在的文件的，而伪目标不需要满足这个要求，同时相应的，make 也不会去检查伪目标的时间戳，而是直接调用他的生成规则，比如：

```

.PHONY: hello

hello:
  touch hello
  echo hello world
```

在这个 makefile 中，无论你敲几次`make hello`，`touch hello`和`echo hello world`这两条生成规则都会被执行。

1.2.2 通用匹配% 和生成规则简化

在上文提到的 Makefile 中，你会发现在编写 a.o 和 b.o 的生成规则的时候，你需要把`build/`和`a.o`这样的多余的东西再写一遍，这个信息明明已经包含在依赖目标和生成目标里了，为了简化规则的编写，makefile 定义了一系列的简写符号，用这些符号我们可以将 Makefile 简化成：

```
.PHONY: binary

binary: build/binary

build/a.o: src/a.c
  mkdir -p $(@D)
  g++ $^ -o $@

build/b.o: b.c
  mkdir -p $(@D)
  g++ $^ -o $@

build/binary: build/a.o build/b.o
  cat $^ > $@
```

其中`$^`出现在生成规则中表示所有被依赖的目标(示例中`$^`会被替换成`build/a.o build/b.o`),`$@`表示生成的目标(示例中上下两条生成规则里的`$@`分别会被替换成`build/a.o`和`build/b.o`)，`$(@D)`则是表示生成目标所在的目录，也就是`build/`，所以上面的 Makefile 其实与之前的 Makefile 完全等价。

符号	含义
`$(@D)`	当前规则目标所在目录
`$@`	当前规则目标
`$<`	当前规则依赖项的第一个
`$^`	当前规则的所有依赖项

表 1: 规则匹配中常见符号

到这里，你会发现一件更撒刁的事，那就是`mkdir -p $(@D)`和`g++ $^ -o $@`这两条语句你在两个目标的生成规则中写了两遍，为了解决这样的问题，Makefile 又提供了通配符机制，利用通配符，我们可以将 Makefile 进一步改写成：

```
.PHONY: binary

binary: build/binary

build/%.o: src/%.c
  mkdir -p $(@D)
```

```
  g++ $^ -o $@

build/binary: build/a.o build/b.o
  cat $^ > $@
```

在这个 Makefile 上，如果我们敲`make binary`，make 会去查找`build/binary`所依赖的`build/a.o`和`build/b.o`的生成规则，当然 make 是找不到的，因为我们并没有明确的写这样的规则。但是 make 会进一步发现，它只要把`build/%.o`中的`%`这个字符替换成`a`，他就能匹配上`build/%.o: src/%.c`这条规则，于是它就把这条规则里面所有的`%`都替换成了`a`，便得到了`build/a.o`的生成规则，同理它也能获得`build/b.o`的生成规则。

1.2.3 变量和函数的使用

在上面的 Makefile 里面，我们还有着这样一个问题，就是如果我们不知道 src 下面究竟有多少`.c`文件，但是我们依旧希望 Makefile 能工作应该怎么办。举个简单的场景，你的项目正在如火如荼的发展中，每天都可能添加若干个`.c`文件或重命名若干个`.c`文件，如果你每次都要手动更新 Makefile，未免太过撒刁了，为了解决这个问题，Makefile 提供了函数机制。

回顾我们的需求，我们其实需要的是 make 能自动找到 src 下所有的.c 文件，并将其重命名为 build 下的.o 文件，最后放到`build/binary`的依赖目标中去：

```
.PHONY: binary

binary: build/binary

build/%.o: src/%.c
  mkdir -p $(@D)
  g++ $^ -o $@

build/binary: $(patsubst src/%.c,build/%.o,$(shell find src/ -name "*.c"))
  cat $^ > $@
```

新的 Makefile 中`build/binary`所依赖的目标已经被替换成了`$(patsubst src/%.c, build/%.o,$(shell find src/ -name "\*.c"))`，我们分开来讲这条语句，首先是`$( shell find src/ -name "\*.c")`，这条语句会调用你终端中的`find`命令，查找`src`目录下所有的`.c`文件，并将其替换成查找的结果，也就是说这条语句执行完毕之后原语句会变成`$( patsubst src/%.c,build/%.o,src/a.c src/b.c)`，`patsubst`是 make 的一个内置函数，它的作用是根据你指定的初始特征和结果特征，对你的输入进入转化，对应到这条语句，你给它的初始特征是`src/%.c`，结果特征是`build/%.o`，输入是`src/a.c src/b.c`，对于输入中的每一项，如`src/a.c`，`patsubst`函数会发现将`%`换成`a`就能匹配上，然后他讲结果特征中的`%`也换成`a`，并将结果替换掉`src/a.c`，也就是说，`src/a.c`会被替换成`build/a.o`，同理`src/b.c`也会被替换成`build/b.o`。

上述 Makefile 看着非常蛋疼，因为我们写了一个超长的语句，有没有一种机制，像编程语言一样，通过定义一些中间变量来拆分超长表达式呢？答案当然是有的，所以上面的 Makefile 又

可以改成下面这样：

```
.PHONY: binary

SRC_DIR := src/
SRCS    := $(shell find $(SRC_DIR) -name "*.o")
OBJ_DIR := build/
OBJS    := $(patsubst src/%.c,build/%.o,$(SRCS))

binary: $(OBJ_DIR)binary

$(OBJ_DIR)%.o: $(SRC_DIR)%.c
  mkdir -p $(@D)
  g++ $^ -o $@

$(OBJ_DIR)binary: $(OBJS)
  cat $^ > $@
```

上面的赋值使用了:=，你暂时不需要知道为什么是:=而不是=，如果你真的想了解，在 docs 目录下的 Makefile.pdf 可以给你答案，同时如果哪天你需要什么其它的功能，你也可以通过查阅Makefile.pdf来看看是否有相应的函数。

函数	含义 (具体用法请查询手册)
$(shell xxx)	执行 shell 命令，并将结果替换到当前位置
$(dir ...)	获取目录
$(notdir ...)	获取文件名
$(realpath ...)	将相对路径转化为绝对路径
$(foreach v, ..., ...)	遍历
$(eval ...)	根据输入的字符串生成新规则
$(subst , , ...)	子串替换
$(patsubst , ..., ...)	基于特征的子串替换

表 2: Makefile 常用函数

1.2.4 宏的使用与自动规则生成

假如你想把你的 Makefile 包装成一个功能库，其他人只要 include 你的 Makefile 就能获得上述所有的功能，那么你需要怎么做呢？一个简单的答案就是宏。

待续。。。

1.3 verilator

verilator 可以将我们的 verilog 代码使用 C++ 模拟，从而使用 C++ 语言可以编写测试程序。verilator 将我们的 verilog 代码编译成一个类，我们只需要在我们的测试的.c 文件中 include 其生成的头文件即可。

1.3.1 前置说明

verilator 的用法，在样例工程中均有体现，建议从样例工程入手，辅之以手册博客，样例项目参见examples/1.Makefile-chisel-sample。

1.3.2 使用 makefile 组织 verilator

从样例项目出发, 组织一个工程的 makefile, 首先得知道这个工程的编译流程以及需要的项目目录的组织 (将源代码和生成的代码放置在不同的文件夹中), 其次将编译过程中的重要的中间文件确定出来, 将其固定命名, 这样可以使得 makefile 的流程更加简单易维护, 然后按照这些中间文件的编译顺序来编写 makefile, 确定依赖, 使用知道的编译命令逐步编译.

1.3.3 Makefile 代码解析

```
.PHONY: xxx
```

.PHONY 定义的伪目标的作用是能够使得 makefile 在每次 make 的时候将这个目标下的指令重新做一遍, 因为当依赖没有改变时,makefile 会默认这个过程不需要再执行一次, 但有时候我们需要这个目标作为一个常用的调用执行, 这样可以将它定义为伪目标

```
SCALA_FILES := $(shell find $(SCALA_DIR) -name "*.scala")
```

这个指令的作用是将 **SCALA_DIR** 目录下寻找所有的 **scala** 后缀的文件.

```
$(EMU_TOP_V): $(SCALA_FILES)
  @mkdir -p $(@D)
  @sbt "run MainDriver -tn $(EMU_TOP_MODULE) -td $(@D) --output-file $@"
```

这一段是将 SCALA 文件使用 sbt 编译成对应的.v 文件, 前面的 @ 标记会使得这条命令不出现在命令行中.

```
$(EMU_MK): $(EMU_TOP_V) $(EMU_CXXFILES)
  @verilator --cc --exe --top-module $(EMU_TOP_MODULE) \
    -o $(notdir $(EMU_BIN)) -Mdir $(@D) \
    --prefix $(basename $(notdir $(EMU_MK))) $^
```

这一段将之前生成的.v 文件和我们编写的 cpp 文件一起编译成 verilator 的 bin 文件, –cc 是说明使用 C++, –exe 是说明和 cpp 一起编译生成一个可执行工程, –top-module 是用于指定出需要编译的顶层模块. -o 指定最后生成的可执行文件名, -Mdir 指明 verilator 生成文件所在目录, –prefix 指明使用的.mk 文件.

```
$(EMU_BIN): $(EMU_MK) $(EMU_CXXFILES)
  @cd $(@D) && make -s -f $(notdir $<)
```

这一段将 verilator 工具生成的代码使用其规则来 make 出最后的可执行文件.

1.3.4 emu/main.cpp 解释

```
#include "emu.h"
#include <memory>
#include <iostream>
#include <verilated.h>

int main(int argc, char **argv) {

/*emu是verilator工具为我们的verilog代码生成的一个类。*/
  auto dut = std::make_shared<emu>();

/*dut->reset表示的是我们的verilog代码中的一个输入信号。*/

  dut->reset = 0;
  for (int i = 0; i < 10; i++) {
    dut->io_in_valid = 1;
    dut->io_in_bits_op = i;
    dut->io_in_bits_a = 123;
    dut->io_in_bits_b = 456;

/*以下4句代码表示了一个周期的变化，clock置0再置1，即为一个周期的变化*/

    dut->clock = 0;
    dut->eval();                  //更新类中各个信号的信息

    dut->clock = 1;
    dut->eval();

/*打印出电路模拟的时候的某个信号的信息，可以写相关代码自动判断信号是否出错*/
  printf ("RECEIVE: %d\n", dut->io_out_bits_c);
  }
  return 0;
}
```

1.4 Chisel

chisel 是由伯克利开发的一门硬件构建语言，嵌入在 scala 编程语言中。相比 verilog，chisel 有抽象的数据类型和接口，层次化 + 面向对象 + 功能化构造，可以很简单地实现工程的高度参数化。可以编译生成出 verilog 语言，相当于软件设计语言中的高级语言。

1.4.1 chisel 安装

使用 chisel 语言需要安装 sbt 官方网站：https://www.scala-sbt.org/1.0/docs/Installing-sbt-on-Linux.html

```
$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.
    list.d/sbt.list
$ sudo apt-key adv --keyserver hkps://keyserver.ubuntu.com:443 --recv 2EE0EA64E40A
    89B84B2DF73499E82A75642AC823
$ sudo apt-get update
$ sudo apt-get install sbt
```

1.4.2 用 chisel 编写硬件

```
git clone https://github.com/addrices/chisel-template.git
```

这是使用 chisel 语言的一个简单的例子，即本项目下 examples/Makefile-chisel-sample, 可以直接从上面的仓库获得。我们简单分析一下这个项目中的各个文件的作用：

这个简单的项目是使用 chisel 编写硬件并且使用 verilator 来进行测试。Makefile 文件中命令的详细解释放置在 verilator.md 中。

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}

/*类似于c语言中的函数，可以在电路重复的地方复用*/
object GTimer {
  def apply(): UInt = {
    val (t, c) = Counter(true.B, 0x7fffffff)
    t
  }
}

/*类似于c语言中的结构体，将相关的信号线打包成一个结构体，其中的Output表示输出（
    Input是输入）*/
class ALU_IN extends Bundle {
  val op = Output(UInt(4.W))
  val b = Output(UInt(32.W))
  val a = Output(UInt(32.W))
}

class ALU_OUT extends Bundle {
  val c = Output(UInt(32.W))
}

class ALU extends Module {
    //表示当前ALU模块的输入输出的信号定义
  val io = IO(new Bundle {
   //这里的Flipped表示的是该结构体中所有Input信号变成Output信号，Output信号变成
       Input信号
    val in = Flipped(ValidIO(new ALU_IN))
```

```
    val out = ValidIO(new ALU_OUT)
  })

  io.in := DontCare

  /* in is valid at the next cycle of valid io.in */
  val in_valid = RegNext(io.in.valid, init=false.B)
  val in = RegEnable(enable=io.in.valid, next=io.in.bits,
    init=0.U.asTypeOf(io.in.bits))

  val neg_b = Mux(in.op(3) === 0.U, in.b, ~in.b + 1.U)
  val sr = Mux(in.op(3) === 0.U,
    in.a >> (in.b(4, 0)),
    (in.a.asSInt >> (in.b(4, 0))).asUInt)

  io.out.valid := in_valid
  io.out.bits.c := Mux1H(Seq(
    (in.op(2, 0) === 0.U) -> (in.a + neg_b),
    (in.op(2, 0) === 1.U) -> (in.a << (in.b(4, 0))),
    (in.op(2, 0) === 2.U) -> (in.a.asSInt < in.b.asSInt).asUInt,
    (in.op(2, 0) === 3.U) -> (in.a < in.b).asUInt,
    (in.op(2, 0) === 4.U) -> (in.a ^ in.b),
    (in.op(2, 0) === 5.U) -> (sr),
    (in.op(2, 0) === 6.U) -> (in.a | in.b),
    (in.op(2, 0) === 7.U) -> (in.a & in.b )
  ))

  when (in_valid) {
    printf("CLOCK: %x, op: %x, a: %x, b: %x; c: %x\n",
      GTimer(), in.op, in.a, in.b, io.out.bits.c)
  }
}

//这个函数是为chisel代码生成verilog代码使用的。指定最后生成的顶层模块即可。
object MainDriver extends ChiselFlatSpec {
  def main(args: Array[String]): Unit = {
    chisel3.Driver.execute(args, () => new ALU)
  }
}
```

在根目录下输入

```
$ sbt
... # 一堆sbt的输出
sbt $ run MainDriver -tn [顶层模块] -td [源文件目录] --output-file [目的地址文件]
```

即可获得对应的.v 文件

1.4.3 相关网站

伯克利的 Chisel 教程: https://github.com/ucb-bar/chisel-tutorial

chisel api 查询: https://www.chisel-lang.org/api/latest/index.html

1.5 minicom 的使用方法

minicom 是一款终端串口连接软件，并且 minicom 自己有一套脚本语法，便于在不定延迟的串口上传送指令

1.5.1 基本的使用

将开发板上的 uart 接口和电脑的 usb 接口相连接，此时在你的 ubuntu 的/dev 目录下会多出一个 ttyUSB1 文件，这个文件就是用于连接串口的钥匙

- 连接串口: `sudo minicom -D /dev/ttyUSB1 -b 115200`
  - `-D /dev/ttyUSB1`的含义显而易见，这里的设备文件有时候是`/dev/ttyUSB0`，具体看实际情况
  - `-b 115200`用于指定串口的波特率，可以理解为传数据的频率，115200 就是每秒传 115200 个比特 (注意：上板设置的波特率和 minicom 设置的波特率一定要一样，不然会乱码)
- 开启 escape 控制码: `minicom -c on ...`
  - 一般情形你在终端里`printf "\e[32mHelloWorld\e[0m"`是会出现彩色文字的 (因为这里使用了控制码`\e[32m`)，彩色文字便于在调试的时候快速定位关键信息，但是 minicom 从串口接受数据并显示的时候会把彩色 (escape 控制码) 给过滤掉，你可以通过`-c on`开启彩色显示

1.5.2 minicom 脚本

一个你们开始上板时会频繁出现的需求，那就是自动化启动加载过程，因为从上板开始，到你们加载 cpu 完毕，你们需要在串口里敲很多命令，而自动化可以给你们省下很多时间:

一个简单的例子，如果你需要启动 u-boot，然后从网口加载程序到 cpu 上，经常会敲的指令:

```
UBOOT> set serverip 192.168.1.104
UBOOT> set ipaddr 192.168.1.107
UBOOT> set gateway 192.168.1.104
UBOOT> tftpboot 0x82000000 nanos-pal
UBOOT> bootelf -p 0x82000000
```

这几条命令还有一个限制，那就是每次敲的时候都必须等待相应的提示符出现之后才能敲命令，过早的敲会被无效掉，而将这一过程写成脚本就是:

```
START:

gosub WAIT_PROMPT
```

```
send "set serverip 192.168.1.104"

gosub WAIT_PROMPT
send "set ipaddr 192.168.1.107"

gosub WAIT_PROMPT
send "set gateway 192.168.1.104"

gosub WAIT_PROMPT
send "tftpboot 0x82000000 nanos-pal"

gosub WAIT_PROMPT
send "bootelf -p 0x82000000"

exit

WAIT_PROMPT:
expect {
  "UBOOT> " break
  goto FAIL
}
return

FAIL:
```

然后下次你再连接串口的时候,使用指定这个脚本就行了(假设上述内容写入了minicom.script这个文件): `sudo minicom -D /dev/ttyUSB1 -b 115200 -c on -S minicom.script`

1.5.3 minicom 的日志记录功能

有的时候你可以需要让 minicom 把所有的内容都记下来，这个时候你加上一个`-C xx.log`选项就行了，这个选项会将所有内容追加到`xx.log`这个文件末尾。

1.6 ILA 的使用方法

在最坏的情况下，你的 cpu 模拟可以跑，仿真可以跑，唯独上板不能跑，这个时候你便需要在上板的时候采样一些信号，来帮助你判断上板运行的状态以及进一步的调试。

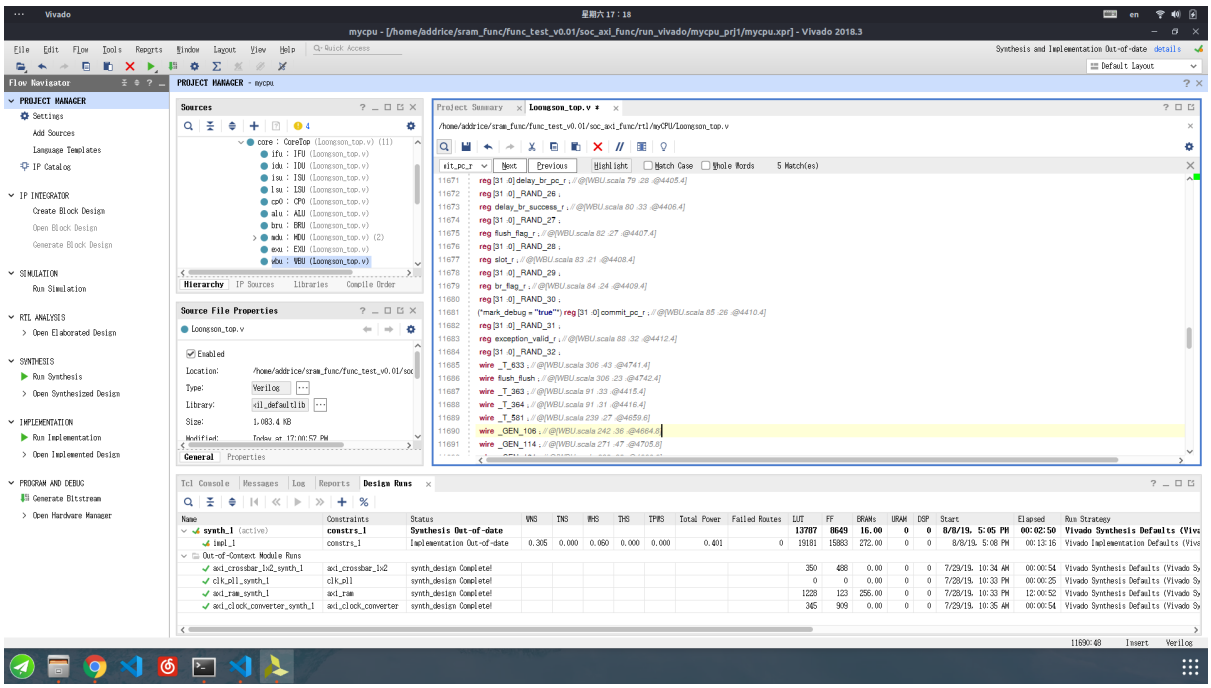
有 2 中方法可以设置 ILA，一种是在.v 文件中生成，第二种是使用 block design 设置。

1.6.1 代码设置

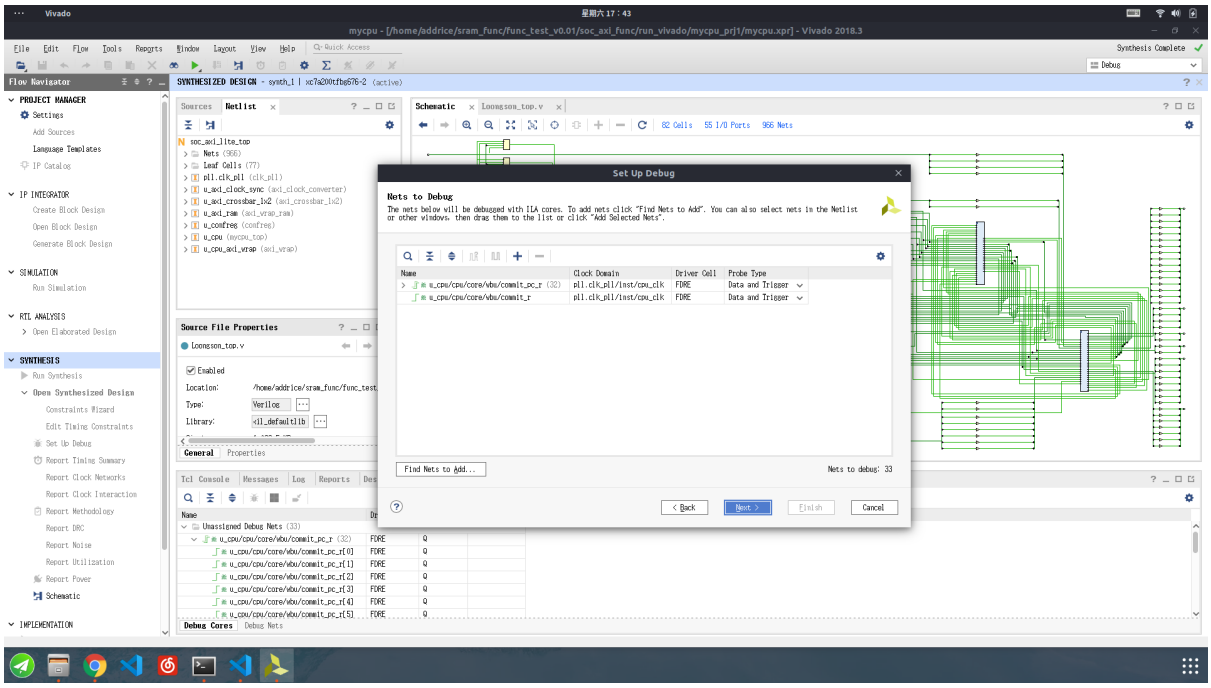
在你需要采样的信号的定义前加上

```
(*mark_debug = "true"*)
```

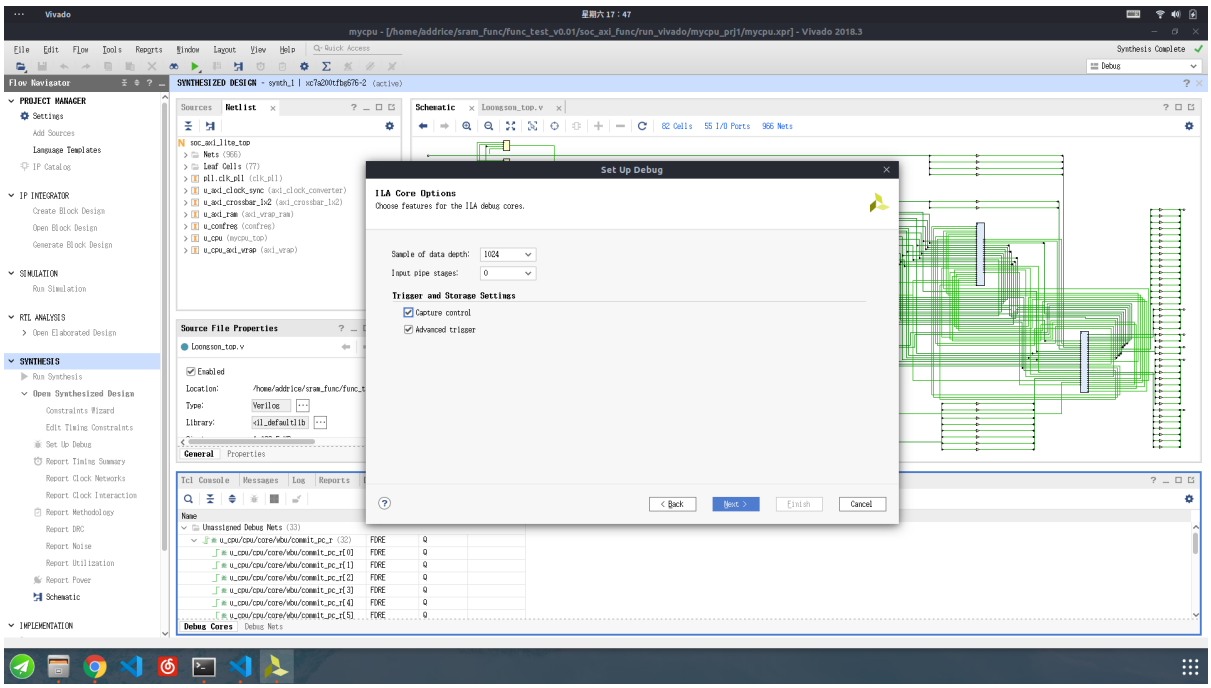
如图所示：



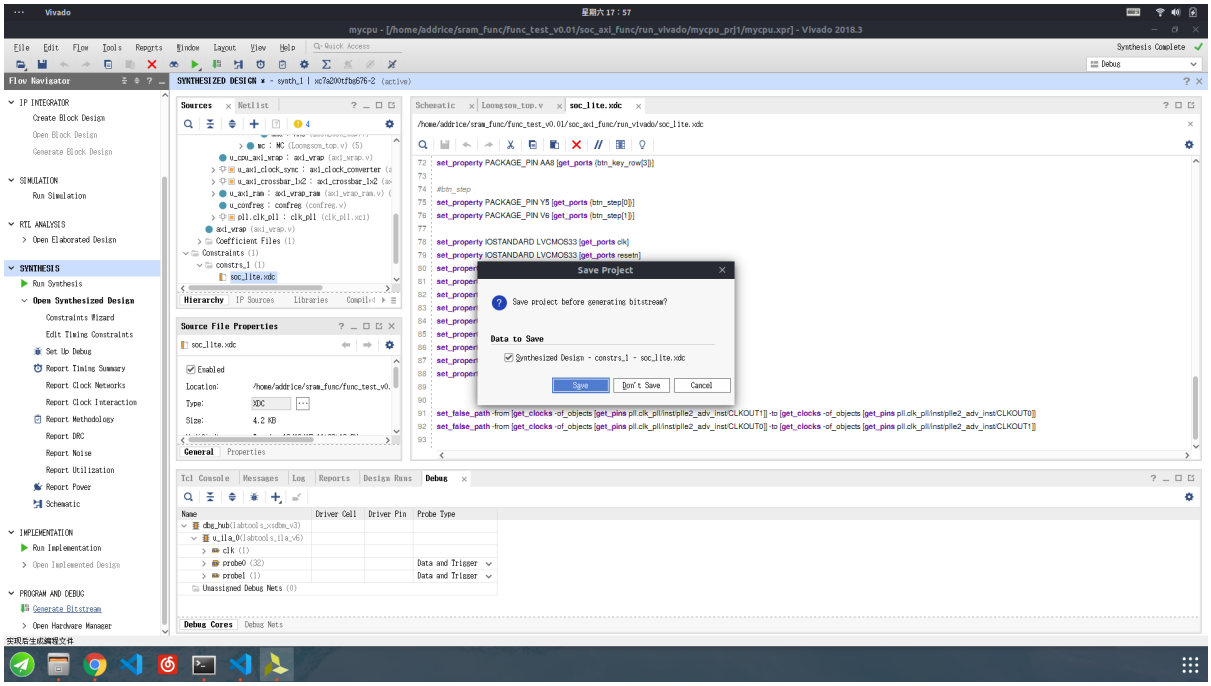
然后综合 (Run synthesis), 完成后 open Synthesized Design, 点击 Set up Debug，如图所示



设置过的 mark_debug 的信号会显示在其中，点击 next，勾选 Captrue control 和 Advanced trigger，如图所示



然后一路 next 完成。这一操作会修改我们的引脚文件，点击生成 bitstream，save 我们对综合文件的修改，如图所示



等待 bitstream 生成完毕上板即可。

1.6.2 block design 上使用 system ila

后来者待续

2 MIPS 相关

2.1 nju-mips 简介

2.1.1 CPU Core

- 单周期 CPU
- 多周期 CPU(考虑信号的阻塞)
- 一般流水线 CPU（标准 5 段、score board）
  - 长延迟指令 (访存，乘除法)
- 高级流水线 CPU (分支预测器、ICache、DCache)
  - CP0 (中断、异常)
  - L2 cache
  - TLB
- 乱序流水线 CPU（rename table、issue queue、ls queue、rob）
- 乱序双发射 CPU

2.1.2 MIPS standard

参见手册 ([1], [2], [3])

- ALU: 算术运算指令 (确定延迟)
  - add, addu, addi, addiu, sub, subu, and, andi, clz, or, ori, xor, xori, nor, slt, sltu, slti, sltiu, sll, sllv, srl, srlv, sra, srav, lui, movn, movz
  - 注意：clz 有高效的递归，采用低效方案容易形成关键路径
- BRU: 分支跳转指令
  - beq, bgtz, blez, bltz, bgez, bltzal, bgezal, bne, j, jal, jalr, jr
  - 注意：所有的分支指令都有延迟槽，即分支后一条指令一定执行
- LSU: 访存指令 (不定延迟)
  - 对齐访存指令: lb, lbu, lh, lhu, lw, sb, sh, sw
  - 不对齐访存指令: lwl, lwr, swl, swr
  - 原子性读写指令: ll, sc
- MDU: 乘除法指令 (长延迟)
  - mul, mult, multu, div, divu, mfhi, mflo, mthi, mtlo
  - 建议：hi 和 lo 寄存器随通用寄存器一起进行转发，如果单独放在 MDU 里面，后续高级流水线的时候需要两套处理方案
- PRU: 特权指令 (该单元又名 CP0)，重点看[3]
  - syscall: 发起系统调用的指令
  - eret: 系统调用执行完从内核态返回用户态的指令

  - mfc0: cp0 有自己的一套标准规定的寄存器，这条指令用于读 cp0 寄存器到一般寄存器
  - mtc0: 用于将一般寄存器的值写入 cp0
  - tlbp: 用于查询一个虚拟地址的页表项是否存在
  - tlbr: 用于读取一个页表项 (cp0 维护虚拟地址到物理地址的映射, 形式有标准规定)
  - tlbw: 用于写入一个页表项
  - tlbwr: 由硬件随机淘汰一个旧页表项，并将新表项写入
  - cache: 用于控制 cache，包括将 cache 的一行写回，标记为无效等操作
  - pref: 内存预取指令，用于加速，具体实现的时候可以什么都不做
  - sync: 多核之间同步的指令，单核可以什么都不做
  - break: 抛异常就行了
  - tlt, tge, tltu, tgeu, tlti, teqi, tgei, tnei, tltiu, tgeiu, tne, teq: 自陷指令，条件满足的时候让 cpu 停住，不要抛异常了，直接在开发板上点个灯，正常执行是不会有这条指令的
- CP0: 0 号协处理器
  - 维护页目录，页目录缺失由操作系统回填，硬件抛异常就行了
  - 维护 cp0 寄存器，以优先级排序：
    * status, cause, epc (实现系统调用必须)
    * badvaddr (实现异常必须，用于指示最后一个出错的访存地址)
    * index, pagemask, context, entry_lo0, entry_lo1, entry_hi (实现虚存必须)
    * count, compare (实现时间中断必须)
    * prid, config, config1 (运行 u-boot 和 linux 必需)
    * base (可选，旧标准内没有，用于调整异常向量表的基址，缺省为 0xbfc00000)
      · cpu 启动时也是 0xbfc00000，所以异常向量一般需要后期写入
- CP0 寄存器的布局
  - 寻址: (待续)

2.1.3 CPU uncore

- [AXI4]: 最常用的核外总线协议
  - 几乎绝大多数硬件的 IP 核都会使用这个协议
  - AXI4lite, 简化版的 AXI4，但是不支持 burst 传输，一开始可以用这个
- Uartlite (用 axi4-uartlite 的 IP 核就行，无需自行实现)
  - 最简单的串口协议
  - 连接串口的简单 [教程](docs/minicom.md)
- Emaclite (不需要深入理解、用就可以了)
  - 最简单的网口协议
- VGA (用往届的就可以了)
  - 数电课的 VGA 控制器包装成 AXI4 控制接口
- BlockRAM、RAM

  - 片上资源，BlockRAM 是读写均同步，适合用来做 cache
  - RAM，写同步，读异步，适合用来做 register

2.1.4 CPU 调试常用的工具链

- Makefile 1.2
  - 用于将一键化各种功能
- tcl 脚本
  - vivado 的自动化脚本，可以调用 vivado 的功能
  - 如果需要一键创建 vivado 项目，这个是常用的工具
    - vivado 可以将当前项目导出到 tcl 脚本
- block design
  - 用图形界面来连接模块和 IP 核
- verilator 1.3
  - verilog 的模拟器，非常快，可以方便你们回归测试
  - 到后期，用到 IP 核的时候，还需要用 vivado 的仿真工具
  - 样例工程在 ‘examples/1.Makefile-chisel-sample‘中
- chisel
  - 高级硬件描述语言
  - 样例工程在 ‘examples/1.Makefile-chisel-sample‘中
- ILA
  - 调试的终极大招，可以在板上电路中采样信号

2.2 CLZ 的递归高效实现

2.2.1 函数版本

```
object CountLeadingZeros32 {
  def apply(in: UInt):UInt = {
    val out = Wire(Vec(5, Bool()))

    out(4) := in(31, 16) === 0.U(16.W)

    val val16 = Mux(out(4), in(15, 0), in(31, 16))
    out(3) := val16(15, 8) === 0.U(8.W)

    val val8  = Mux(out(3), val16(7, 0), val16(15, 8))
    out(2) := val8(7, 4) === 0.U(4.W)

    val val4  = Mux(out(2), val8(3, 0), val8(7, 4))
    out(1) := val4(3, 2) === 0.U(2.W)
```

```
    out(0) := Mux(out(1), ~val4(1), ~val4(3))

    Mux(in === 0.U, 32.U, out.asUInt)
  }
}
```

2.2.2 模块版本 (不建议使用)

```
class CountLeadingZeros32 extends Module {
  val io = IO(new Bundle {
  val in = Input(UInt(32.W))
  val out = Input(UInt(32.W))
  })

  val tmp = Wire(Vec(5, Bool()))

  tmp(4) := io.in(31, 16) === 0.U(16.W)

  val val16 = Mux(tmp(4), io.in(15, 0), io.in(31, 16))
  tmp(3) := val16(15, 8) === 0.U(8.W)

  val val8  = Mux(tmp(3), val16(7, 0), val16(15, 8))
  tmp(2) := val8(7, 4) === 0.U(4.W)

  val val4  = Mux(tmp(2), val8(3, 0), val8(7, 4))
  tmp(1) := val4(3, 2) === 0.U(2.W)

  tmp(0) := Mux(tmp(1), ~val4(1), ~val4(3))

  io.out := Mux(io.in === 0.U, 32.U, tmp.asUInt)
}
```

3 相关项目说明

4 运行及移植 linux

参考文献