

计算机系统综合实验手册

目录

1	基本工具链	4
1.1	git 基本用法	4
1.2	makefile	5
1.2.1	规则依赖	6
1.2.2	通用匹配% 和生成规则简化	8
1.2.3	变量和函数的使用	9
1.2.4	宏的使用与自动规则生成	10
1.3	verilator	10
1.3.1	前置说明	11
1.3.2	使用 makefile 组织 verilator	11
1.3.3	Makefile 代码解析	11
1.3.4	emu/main.cpp 解释	12
1.4	chisel	12
1.4.1	chisel 安装	12
1.4.2	用 chisel 编写硬件	13
1.4.3	相关网站	15
1.5	minicom 的使用方法	15
1.5.1	基本的使用	15
1.5.2	minicom 脚本	15
1.5.3	minicom 的日志记录功能	16
1.6	ILA 的使用方法	16
1.6.1	代码设置	16
1.6.2	block design 上使用 system ila	18
2	MIPS 相关	21
2.1	nju-mips 简介	21
2.1.1	CPU Core	21
2.1.2	MIPS standard	21
2.1.3	CPU uncore	22
2.1.4	SOC 构建与调试常用的工具链	23
2.2	CLZ 的递归高效实现	23
2.2.1	函数版本	23
2.2.2	模块版本 (不建议使用)	24

3	相关项目说明	25
3.1	测试相关	25
3.1.1	insttest	25
3.1.2	tlbtest	25
3.1.3	nexus-am/tests/cputest	25
3.1.4	nexus-am/apps/coremark	25
3.1.5	nexus-am/apps/microbench	25
3.2	运行 linux 相关	25
3.2.1	u-boot	26
3.2.2	linux	26
3.2.3	buildroot	27
3.3	nemu-mips32	28
4	运行及移植 linux	30
4.1	运行 linux	30

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:ORKaY5n3N4G0P4guAbT0UFNZNorA3mqSL0S0xPW0/+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
<code>\$(@D)</code>	当前规则目标所在目录
<code>\$@</code>	当前规则目标
<code>\$<</code>	当前规则依赖项的第一个
<code>\$^</code>	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，我们实际需要的是 `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，对你的输入进行转化，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
\$(abspath ...)	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 htps://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, 便于在不用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, 否则会乱码)
- 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。

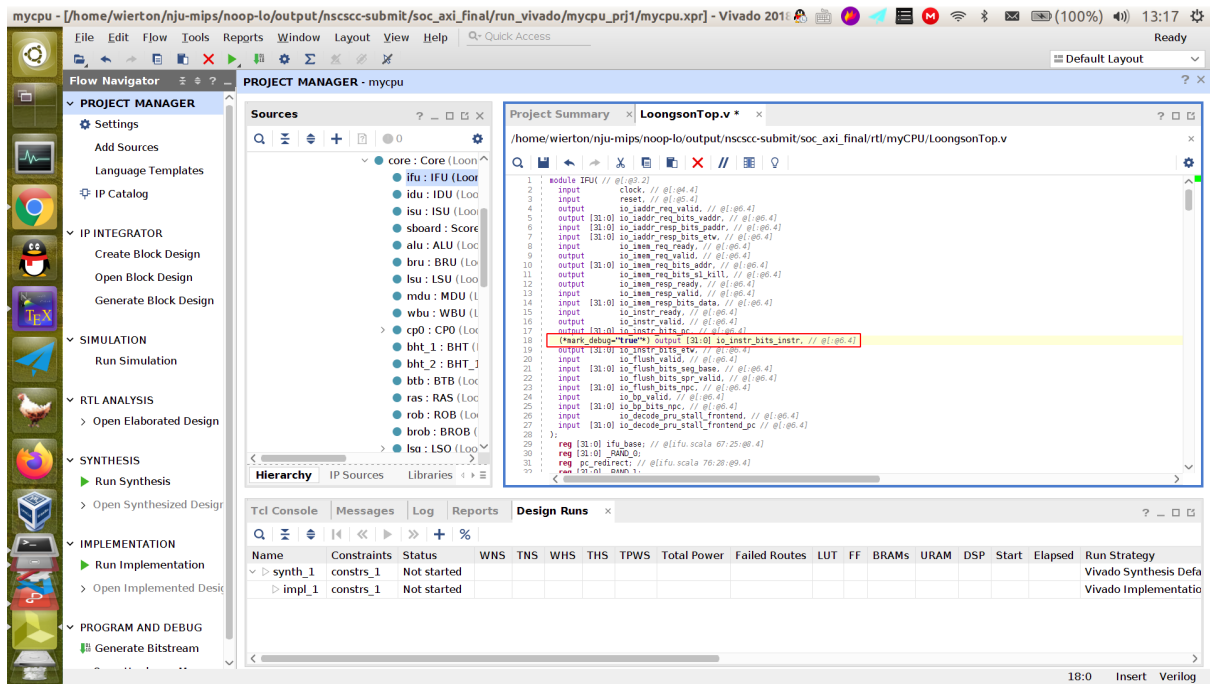
有 2 中方法可以设置 ILA,一种是在 `.v` 文件中生成,第二种是使用 `block design` 设置。

1.6.1 代码设置

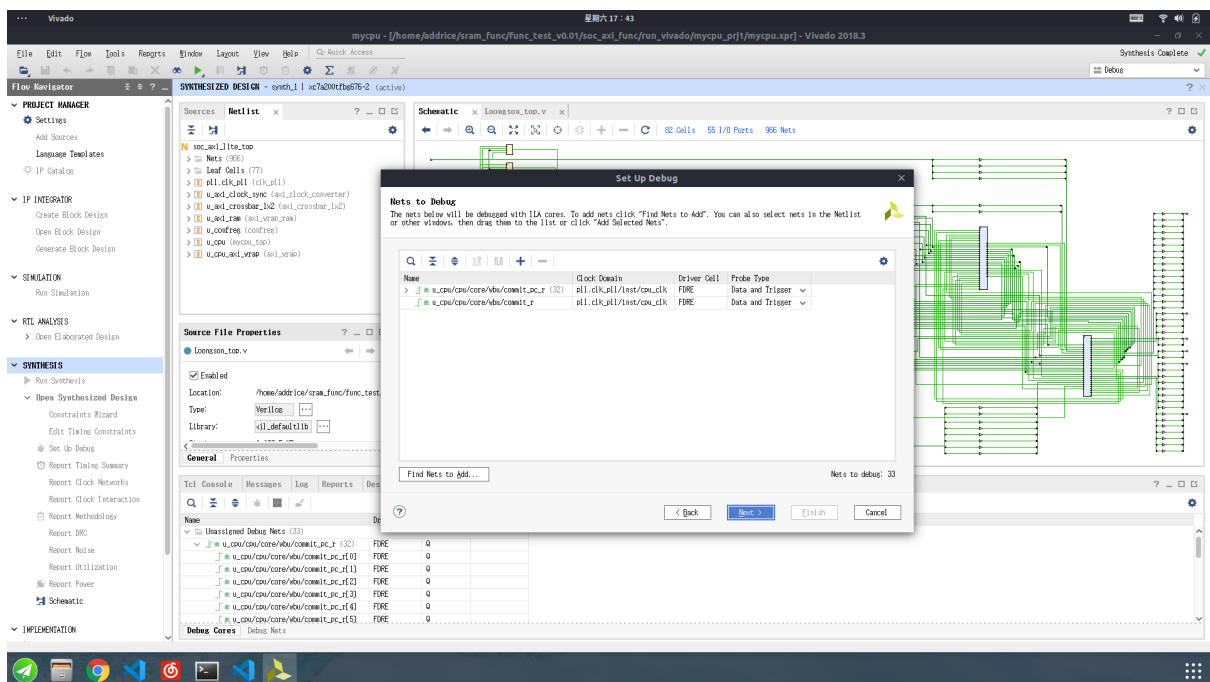
在你需要采样的信号的定义前加上

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

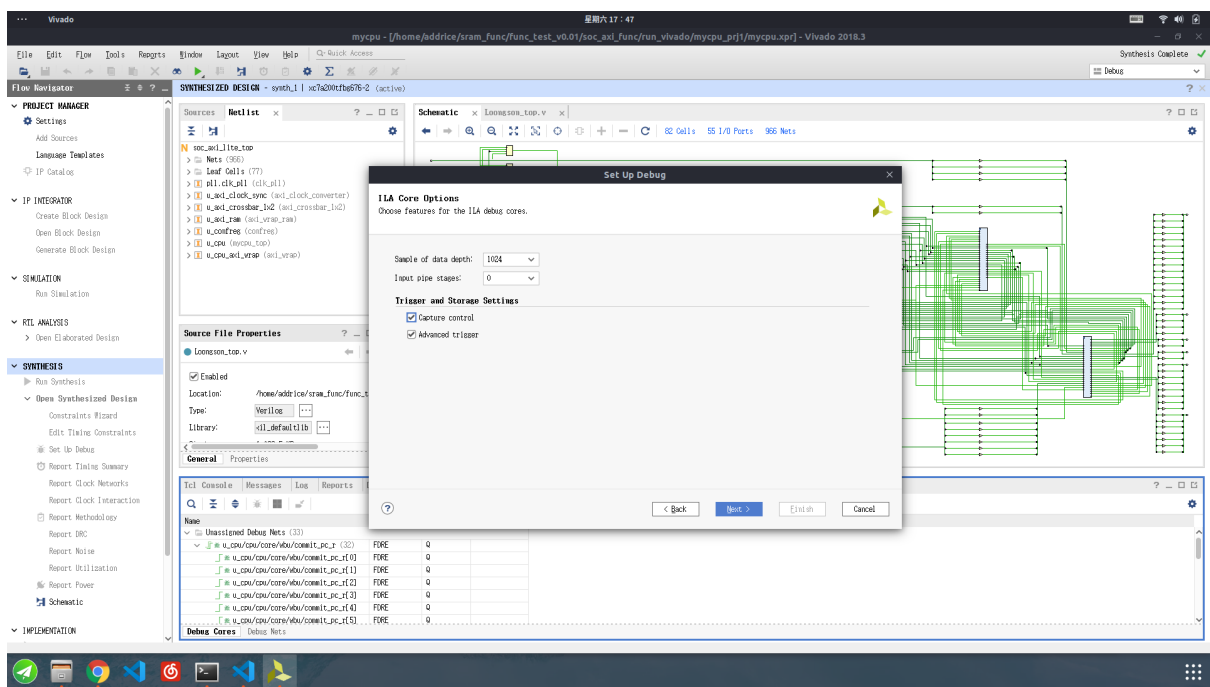
如图所示:



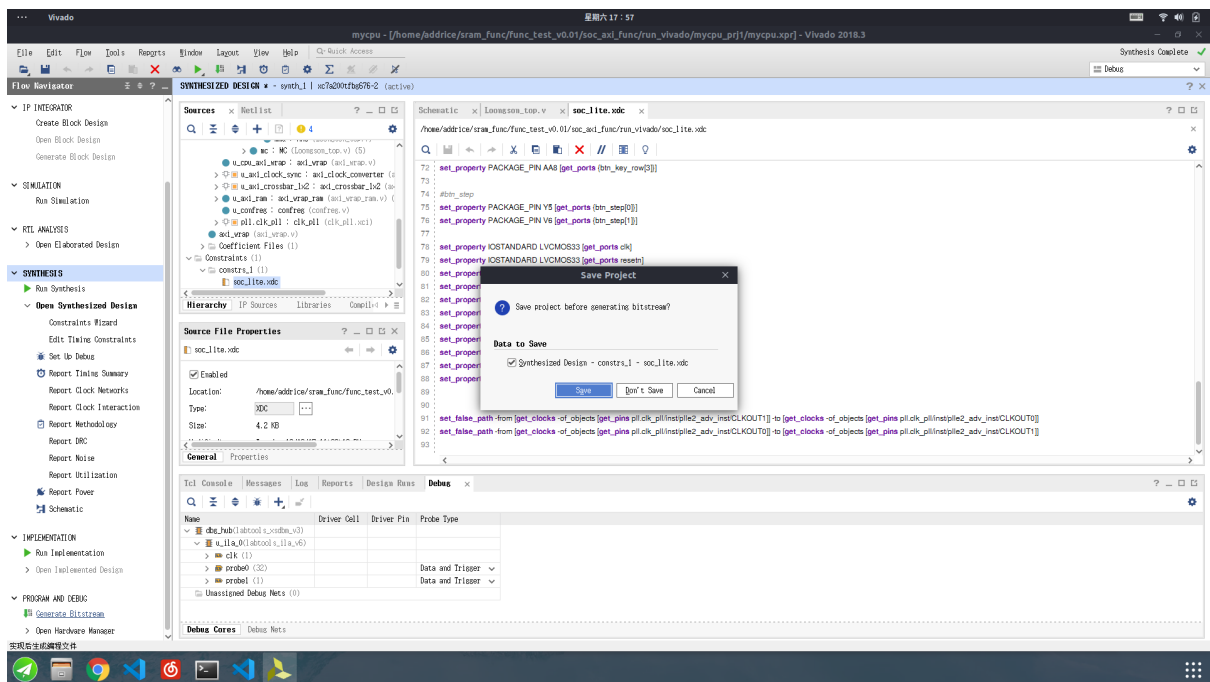
然后综合 (Run synthesis), 完成后 open Synthesized Design, 点击 Set up Debug, 如图所示



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



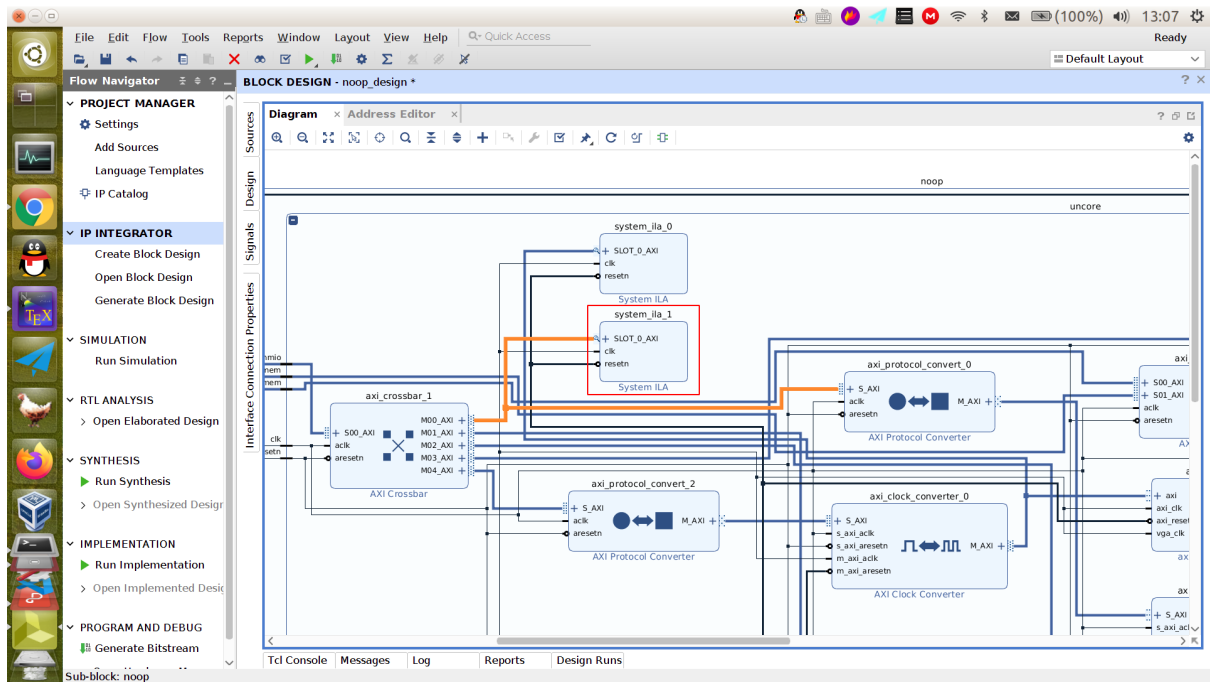
然后一路 next 完成。这一操作会修改我们的引脚文件，点击生成 bitstream，save 我们对综合文件的修改，如图所示



等待 bitstream 生成完毕上板即可。

1.6.2 block design 上使用 system ila

system ila 用来抓取 AXI4 的信号，相较于用 (*mark_debug = "true"*), system ila 抓取的信号易读性更强，并且可以直接在 block design 中操作。如下图所示，首先打开 block design:



后续流程后前面一样，综合上板，然后便会自动进入抓信号的界面。

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，下述以建议的实现顺序排序：
 - * 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)
- CP1: 1 号协处理器
 - 关于浮点数的，可以使用 IP 核，需要实现的指令数不多，可在后续提升时实现

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[1.5](#)
- Emaclite (不需要深入理解、用就可以了)
 - 最简单的网口协议，vivado 里面有 IP 核，u-boot 和 linux 里面均有驱动
- SPI flash

- 注意 SPI 仅为通信协议，实际想要读写 flash，需通过 SPI 的 IP 核将命令发送给 flash 芯片，关于 flash 的命令，这里不推荐任何手册。其状态机过于复杂，这里不建议花太多时间
- 如果想要使用 flash，请先将 linux 跑起来，linux 里有驱动，无需再自己实现
- PS/2
 - 注意，PS/2 也仅是通信协议，用于与键盘和鼠标交互，实际读取键盘和鼠标还需基于 PS/2 发送命令
 - PS/2 在 vivado 里面有 IP 核，在 linux 里面有驱动，如果可以跑通 linux，那么使用 PS/2 是比较容易的
- 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 SOC 构建与调试常用的工具链

- 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

写在前面，所有的相关项目只做概述性说明。由于项目本身也在经常重构，一些整理的内容到后期可能就不对了，同时对于计算机系统综合实验的同学而言，看代码也是一个必由之路。

3.1 测试相关

3.1.1 insttest

这个项目是从龙芯的测试集和清华的测试集里扒出来的，用于进行单指令测试。需要注意的一点是，一旦开始流水线设计，单指令测试只能验证最基本的正确性，由于流水线的指令间并行性，不同的指令流可能会导致不同的状态机切换，因而造成单指令测试所无法触发的 bug。

3.1.2 tlbttest

这个是从龙芯的测试集中扒出来的，用于进行 tlbt 的测试，需要注意的是，这个测试并不完备，完备的测试还是需要运行 linux 这样的 workload。

3.1.3 nexus-am/tests/cputest

基于 AM 的一些简单测试，继承自 PA，可以测试一些简单的 workload

3.1.4 nexus-am/apps/coremark

基于 AM 的评分程序，workload 中等。

3.1.5 nexus-am/apps/microbench

基于 AM 的大型 workload，很多流水线 bug 可以在这个测试下暴露出来，需要予以重视。这个测试由于运行时间过长，大部分时候可能只能在板上测试，届时可能需要大量使用 ILA。一般普通流水线的 microbench 跑分可以到达 200-300 分，这个成绩大概是决赛的垫底成绩，如果想进龙芯杯决赛，还需要多多优化。

3.2 运行 linux 相关

运行 linux 相关的几个项目都使用了 Kconfig 进行配置，在 nju-mips 的几个仓库里面，均已经准备好了针对 noop 的配置文件，在编译前通过 `make noop_defconfig` 即可初始化配置，除此以外还可以通过 `make menuconfig` 来进行手动配置。后期可以肯定还需要对 linux 进行裁剪适配，手动配置应是必须掌握的技能之一。

除了 Kconfig, u-boot 和 linux 还使用 dts 文件来描述 soc 所能访问的外设。dts 文件的语法可以百度，如果了解某个外设的 dts 文件怎么写，可以查询 xilinx 的官网。比如加一个 emaclite 硬件，

可以访问<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842124/U-Boot+Ethernet+Driver>查看具体流程

3.2.1 u-boot

u-boot 是一个引导程序，一般建议直接烧录在 SOC 的 sram 中，用于引导程序。其正确运行依赖于 cp0 里面的 cache 指令。u-boot 中包含丰富的驱动，结合 vivado 丰富的 IP 核，使得我们可以简单组合一下 uncore 和配置一下 u-boot 就能让干成很多事情。比如从网口加载内核并执行。

u-boot 的 (重要) 目录结构如下：

```
u-boot
+-- board/                # 初始化板子的代码
+-- drivers/              # 各种驱动代码
|   +-- serial/
|   |   +-- serial_xuartlite.c # xilinx uartlite驱动，有大小端bug，nju-mips上已修复
|   +-- spi/
|   |   +-- xilinx_spi.c      # spi驱动，与flash相关
|   +-- net/
|       +-- xilinx_emaclite.c # xilinx的emaclite驱动
+-- arch/
|   +-- mips/
|       +-- mach-noop/        # 移植到noop所需目录
|       +-- dts/
|           +-- noop.dts      # 描述noop soc外设的文件
+-- configs/
    +-- noop_defconfig       # noop的配置文件
```

3.2.2 linux

这个项目是 linux 的内核，无需多言。请务必注意的一点是，在 mips 里面异常和异常号不是一一对应的，同一个异常号对应的异常可能实际走的异常入口并不一样，这一点手册没有很好的说明，除此以外如果 linux 运行不起来请着重检查 cp0 和 tlb，尤其是 tlb。

linux 的重要目录结构如下：

```
u-boot
+-- drivers/              # 各种驱动代码
|   +-- tty/
|   |   +-- serial/
|   |       +-- uartlite.c    # uartlite驱动，mainline有bug，已修复
|   +-- spi/
|   |   +-- spi-xilinx.c     # spi驱动，与flash相关
|   +-- net/
|       +-- etherlite/
|       +-- xilinx/
|           +-- xilinx_emaclite.c
|           +-- xilinx_axinet.c
```

```

+-- arch/
  +-- mips/
    +-- noop/                # 移植到noop所需目录
    +-- boot/
      | +-- dts/
      |   +-- noop/
      |       +-- noop.dts # 描述noop soc外设的文件
    +-- configs/
      +-- noop_defconfig    # noop的配置文件

```

关于如何驱动外设,这里需要额外说明一下。假设现在需要驱动 `uartlite` 这个外设,首先需要运行 `make menuconfig` 将这个外设的驱动开启,然后需要进入 `noop.dts` 文件中,添加对 `uartlite` 这个外设的描述,最后在启动时也就有这个外设了。那么这个外设怎么访问呢?还是以 `uartlite` 为例,我们打开其驱动文件 `uartlite.c`,会发现有这样几行:

```

#define ULITE_NAME      "ttyUL"
#define ULITE_MAJOR     204
#define ULITE_MINOR     187

static struct uart_driver ulite_uart_driver = {
    .owner      = THIS_MODULE,
    .driver_name = "uartlite",
    .dev_name   = ULITE_NAME,
    .major      = ULITE_MAJOR,
    .minor      = ULITE_MINOR,
    .nr         = ULITE_NR_UARTS,
#ifdef CONFIG_SERIAL_UARTLITE_CONSOLE
    .cons       = &ulite_console,
#endif
};

```

这几行意味这, `uartlite` 这个外设被检测到以后,会在 `/dev` 下生成 `/dev/ttyUL` 这个设备文件,其对应的设备号是 204, 187, 如果有多个外设,则他们从 0 开始排序,如 `/dev/ttyUL0`、`/dev/ttyUL1`、...

有时候你会在百度的时候发现有些 `linux` 启动的时候会有参数 `root=/dev/mmcblock`,这就意味这有一个外设名为 `mmcblock` 被注册了,并且对应的硬件在 `dts` 文件里面被描述了,在这两个前提下, `linux` 可以找到这个块设备,并将其设为根文件系统。

3.2.3 buildroot

`linux` 的运行依赖于一个根文件系统,这个根文件系统中存放这各种各样的配置文件和程序(和我们的系统一样)。`linux` 启动时会加载根文件系统里的 `/bin/init` 程序作为 1 号进程。`buildroot` 的作用是帮你生成这样一个文件系统。

`buildroot` 会在 `make` 之后去网上下载源代码、编译各种程序、最后构建根文件系统。下载好的源代码会放在 `dl` 目录下,编译的中间结果会放在 `output/build` 目录下,生成的文件系统镜像的

初始目录会在 `output/target` 下，生成的最终镜像文件会在 `output/images` 目录下。

一个标准的启动流程是，`/bin/init` 被启动后去加载 `inittab`，其文件大概如下：

```
# Startup the system
::sysinit:/bin/mount -t proc proc /proc
::sysinit:/bin/mount -o remount,rw /
::sysinit:/bin/mkdir -p /dev/pts /dev/shm
::sysinit:/bin/mount -a
::sysinit:/sbin/swapon -a
null::sysinit:/bin/ln -sf /proc/self/fd /dev/fd
null::sysinit:/bin/ln -sf /proc/self/fd/0 /dev/stdin
null::sysinit:/bin/ln -sf /proc/self/fd/1 /dev/stdout
null::sysinit:/bin/ln -sf /proc/self/fd/2 /dev/stderr
::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
# ttyUL0::askfirst:/bin/login
::sysinit:/etc/init.d/rcS

# Put a getty on the serial port
ttyUL0::respawn:/sbin/getty -L ttyUL0 115200 vt100

# Stuff to do for the 3-finger salute
#::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
::shutdown:/etc/init.d/rcK
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
```

需要额外关注的是 `ttyUL0::respawn:/sbin/getty -L ttyUL0 115200 vt100` 这一行，意思是将中断绑定到 `ttyUL0` 这个串口外设上，设置波特率为 115200。`respawn` 和 `/sbin/getty` 的意思是，`init` 结束后会启动 `/sbin/getty` 这个程序作为 `shell`，并且如果退出 `/sbin/getty` 会重新启动这个程序。`/sbin/getty` 是一个账户管理程序，他会在启动时要求输入账户和密码（均为 `root`），然后再启动 `sh` 作为 `shell`。

3.3 nemu-mips32

这个项目是一个 `mips32` 的模拟器，目前已成功运行 `linux`，具备相对齐全的 `CP0` 和 `TLB`，其整体的大致目录结构如下：

```
nemu-mips32
+-- Kconfig      # 用于配置 nemu-mips32
+-- Makefile
+-- configs/     # 存放已经保存好的各种配置
|   +-- noop_defconfig
|   +-- mips32r1_defconfig
```

```
+-- src/
|   +-- main.c
|   +-- cpu/      # 模拟mips32的核心代码
|   +-- dev/      # 模拟外设的核心代码
|   +-- monitor/  # 用于解析命令行参数和与gdb交互
|   +-- utils/    # 项目所依赖的一些小功能函数
+-- kconfig/      # Kconfig对应的配置工具源码
```

需要注意，由于所涉及的配置繁多，nemu-mips32 采用 Kconfig 对整个项目进行配置，包括各种外设的开闭、各种优化机制的开闭、各种调试机制的开闭、一些功能的开闭等。运行make menuconfig可以对项目进行配置，配置完毕退出会在当前目录下生成.config文件，并在include/下生成 generated/文件夹用于存放配置的头文件。

如果需要阅读 nemu-mips32 的代码，请集中精力在 src/cpu/和 src/dev/这两个目录下，其它代码可以视为干扰项，整个项目的入口在 src/目录下的 main.c 里面。

4 运行及移植 linux

写在前面，linux 目前以基本移植完毕，但如果想在决赛中取得好名次，比如想运行 debian，建议自己摸索一遍 linux 的移植过程。

4.1 运行 linux

首先 clone 项目

```
git clone git@github.com:nju-mips/buildroot
git clone git@github.com:nju-mips/u-boot
git clone git@github.com:nju-mips/linux
```

其次编译各项目，依序

```
cd buildroot
make noop_defconfig
make # buildroot 编译特别耗时，请不到万不得已不要 clean 这个项目

cd u-boot
ARCH=mips CROSS_COMPILE=mips-linux-gnu make noop_defconfig
ARCH=mips CROSS_COMPILE=mips-linux-gnu make -j8

cd linux
ARCH=mips CROSS_COMPILE=mips-linux-gnu make noop_defconfig
ARCH=mips CROSS_COMPILE=mips-linux-gnu make -j8
```

接着是编译 nemu-mips32

```
cd nemu-mips32
make mips32r1_defconfig
make
```

最后是运行：

```
nemu $ sudo build/nemu -b -e ../u-boot/u-boot --block-data ddr:0x4000000:../linux/
      vmlinux
<debug_uart>

U-Boot 2020.01-rc5-ga5281d02 (Feb 06 2020 - 22:32:13 +0800)

Model: uart0
DRAM:  128 MiB
In:     serial@bfe50000
Out:    serial@bfe50000
Err:    serial@bfe50000
Net:    EMACLite: bff00000, phyaddr 1, 1/1
```

```

Warning: ethernet@bfff00000 using MAC address from ROM
eth0: ethernet@bfff00000
noop # bootelf -p 0x84000000
## Starting application at 0x803de6a0 ...
[ 0.000000] Linux version 5.3.0+ (wierton@wierton-ThinkPad-T480s) (gcc version
7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)) #1 Thu Feb 6 22:32:34 CST 2020
[ 0.000000] printk: bootconsole [early0] enabled
[ 0.000000] CPU0 revision is: 00018000 (MIPS 4Kc)
[ 0.000000] MIPS: machine is uart0
[ 0.000000] Determined physical RAM map:
[ 0.000000] memory: 08000000 @ 00000000 (usable)
[ 0.000000] Initrd not found or empty - disabling initrd
[ 0.000000] Primary instruction cache 16kB, VIPT, 4-way, linesize 16 bytes.
[ 0.000000] Primary data cache 16kB, 4-way, VIPT, no aliases, linesize 16 bytes
[ 0.000000] Zone ranges:
[ 0.000000] Normal [mem 0x0000000000000000-0x0000000007ffffff]
[ 0.000000] Movable zone start for each node
[ 0.000000] Early memory node ranges
[ 0.000000] node 0: [mem 0x0000000000000000-0x0000000007ffffff]
[ 0.000000] Initmem setup node 0 [mem 0x0000000000000000-0x0000000007ffffff]
[ 0.000000] Built 1 zonelists, mobility grouping on. Total pages: 32512
[ 0.000000] Kernel command line: cca=0 console=ttyUL0,115200n8 rootfstype=ext4
root=/dev/mtdblock0
[ 0.000000] Dentry cache hash table entries: 16384 (order: 4, 65536 bytes,
linear)
[ 0.000000] Inode-cache hash table entries: 8192 (order: 3, 32768 bytes, linear)
8001ba50: mtc0 $0 , $6, 0
[ 0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[ 0.000000] Memory: 122540K/131072K available (3987K kernel code, 154K rwdata,
748K rodata, 2144K init, 325K bss, 8532K reserved, 0K cma-reserved)
[ 0.000000] SLUB: HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[ 0.000000] NR_IRQS: 128
[ 0.000000] random: get_random_bytes called from start_kernel+0x378/0x570 with
crng_init=0
[ 0.000000] mips_hpt_frequency: 50000000 HZ
[ 0.000000] clocksource: MIPS: mask: 0xffffffff max_cycles: 0xffffffff,
max_idle_ns: 38225208935 ns
[ 0.000067] sched_clock: 32 bits at 50MHz, resolution 20ns, wraps every
42949672950ns
[ 0.001370] Console: colour dummy device 80x25
[ 0.001819] Calibrating delay loop... 76.80 BogoMIPS (lpj=38400)
[ 0.013814] pid_max: default: 32768 minimum: 301
[ 0.015292] Mount-cache hash table entries: 1024 (order: 0, 4096 bytes, linear)
[ 0.015826] Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes,
linear)
[ 0.026179] devtmpfs: initialized

```

```

[ 0.038005] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff,
max_idle_ns: 1911260446275000 ns
[ 0.038565] futex hash table entries: 256 (order: -1, 3072 bytes, linear)
[ 0.039095] pinctrl core: initialized pinctrl subsystem
[ 0.041428] NET: Registered protocol family 16
[ 0.045203] cpuidle: using governor menu
[ 0.177085] kworker/u2:0 (15) used greatest stack depth: 6764 bytes left
[ 0.255692] pps_core: LinuxPPS API ver. 1 registered
[ 0.255947] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
<giometti@linux.it>
[ 0.261277] clocksource: Switched to clocksource MIPS
[ 0.273472] random: fast init done
[ 0.305108] kworker/u2:7 (116) used greatest stack depth: 6388 bytes left
[ 0.439554] NET: Registered protocol family 2
[ 0.444856] tcp_listen_portaddr_hash hash table entries: 512 (order: 0, 4096
bytes, linear)
[ 0.445613] TCP established hash table entries: 1024 (order: 0, 4096 bytes,
linear)
[ 0.446233] TCP bind hash table entries: 1024 (order: 0, 4096 bytes, linear)
[ 0.446837] TCP: Hash tables configured (established 1024 bind 1024)
[ 0.447742] UDP hash table entries: 256 (order: 0, 4096 bytes, linear)
[ 0.448176] UDP-Lite hash table entries: 256 (order: 0, 4096 bytes, linear)
[ 0.449471] NET: Registered protocol family 1
[ 1.507595] workingset: timestamp_bits=30 max_order=15 bucket_order=0
[ 1.579196] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 252)
[ 1.579699] io scheduler kyber registered
[ 1.581658] lfe50000.serial: ttyUL0 at MMIO 0x1fe50000 (irq = 4, base_baud = 0)
is a uartlite
[ 1.582184] printk: console [ttyUL0] enabled
[ 1.582184] printk: console [ttyUL0] enabled
[ 1.582731] printk: bootconsole [early0] disabled
[ 1.582731] printk: bootconsole [early0] disabled
[ 1.677182] m25p80 spi0.0: n25q512a (65536 Kbytes)
[ 1.677605] 1 fixed-partitions partitions found on MTD device spi0.0
[ 1.677912] Creating 1 MTD partitions on "spi0.0":
[ 1.678207] 0x0000000000000-0x0000000800000 : "rootfs"
[ 1.692079] xilinx_spi 1fe80000.spi: at 0x1FE80000 mapped to 0x(ptrval), irq=5
[ 1.695031] libphy: Fixed MDIO Bus: probed
[ 1.695790] xilinx_emaclite 1ff00000.ethernet: Device Tree Probing
[ 1.697052] libphy: Xilinx Emaclite MDIO: probed
[ 1.738520] xilinx_emaclite 1ff00000.ethernet: MAC address is now 08:86:4c:0d:f7
:09
[ 1.744147] xilinx_emaclite 1ff00000.ethernet: Xilinx EmacLite at 0x1FF00000
mapped to 0xBFF00000, irq=6
[ 1.747840] NET: Registered protocol family 17
[ 1.786954] Freeing unused kernel memory: 2144K
[ 1.787371] This architecture does not have kernel memory protection.

```



```
[ 1.787824] Run /init as init process
/bin/sh: can't access tty; job control turned off
/ #
```

缺省状态下 linux 会初始化 spi 和 emaclite，这可以通过修改 linux/arch/mips/boot/dts/noop.dts 文件，将相应的外设注释掉来避免。如果没有在 nemu-mips32 配置开启相应的外设，那么启动的时候会出现如下错误：

```
nemu: src/cpu/cpu.c:221: vaddr_write: Assertion 'dev && dev->write' failed
CPUAssert message: bad addr bfe80060

[1] 24556 abort      sudo build/nemu -b -e ../u-boot/u-boot --block-data
```

这种时候请检查 bad addr 对应的外设，并予以开启。除此以外还需要额外注意的一点是，emaclite 的模拟需要与 tun/tap 进行交互，为此 root 权限是必须的，因此如果要开启 nemu 的虚拟网卡，运行的时候需要加上 sudo。