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
$(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 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。

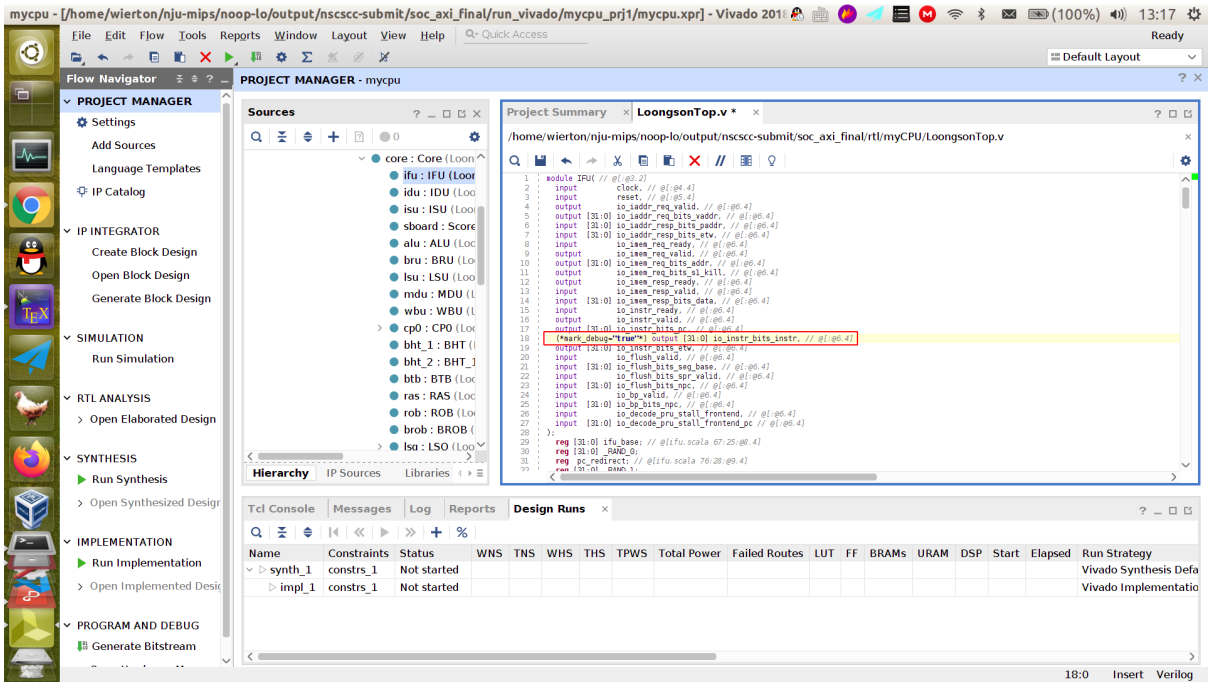
有 2 中方法可以设置 ILA，一种是在.v 文件中生成，第二种是使用 block design 设置。

1.6.1 代码设置

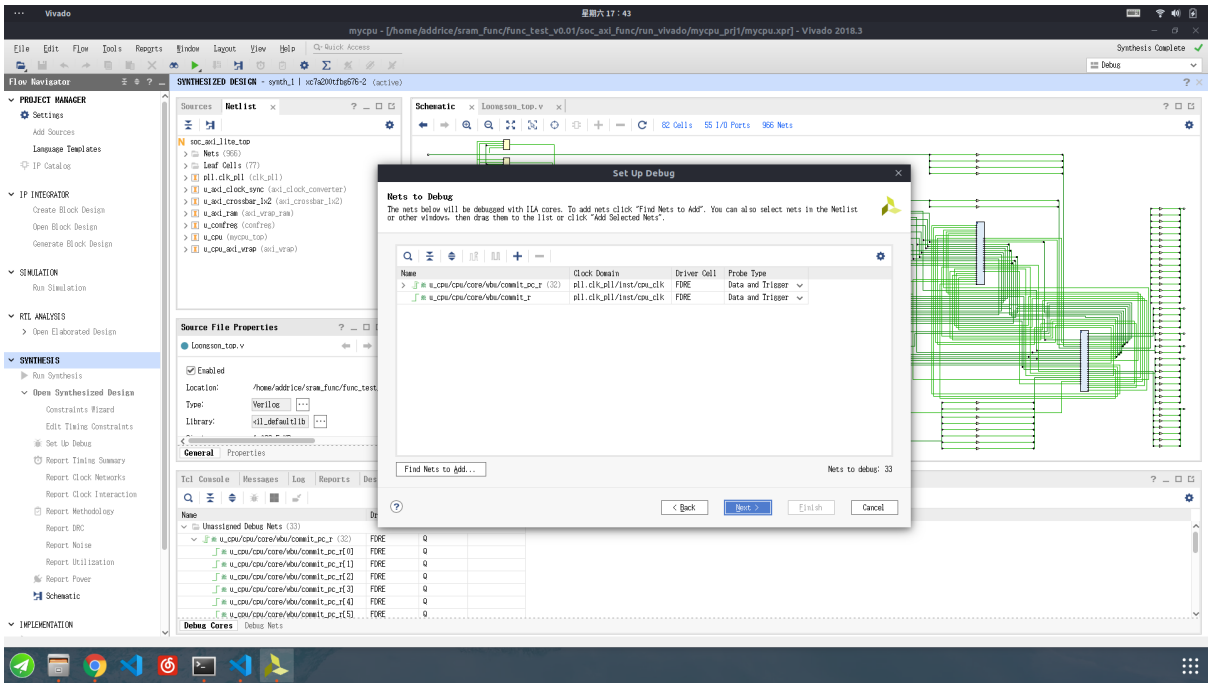
在你需要采样的信号的定义前加上

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

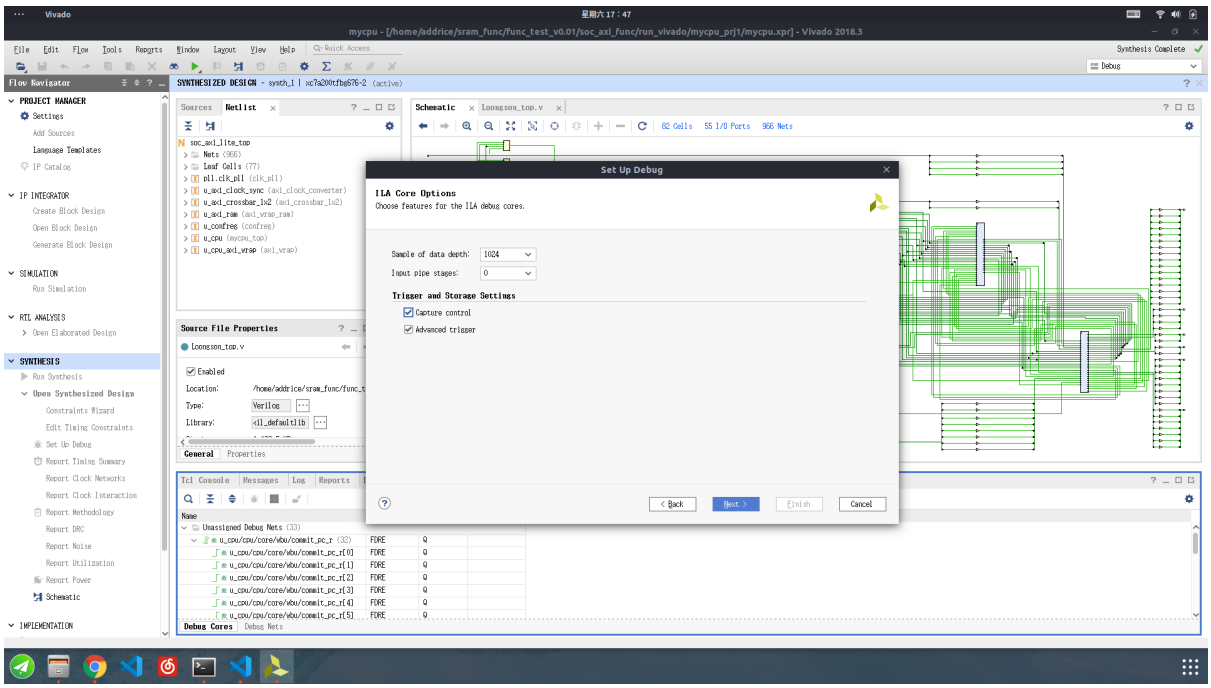
如图所示：



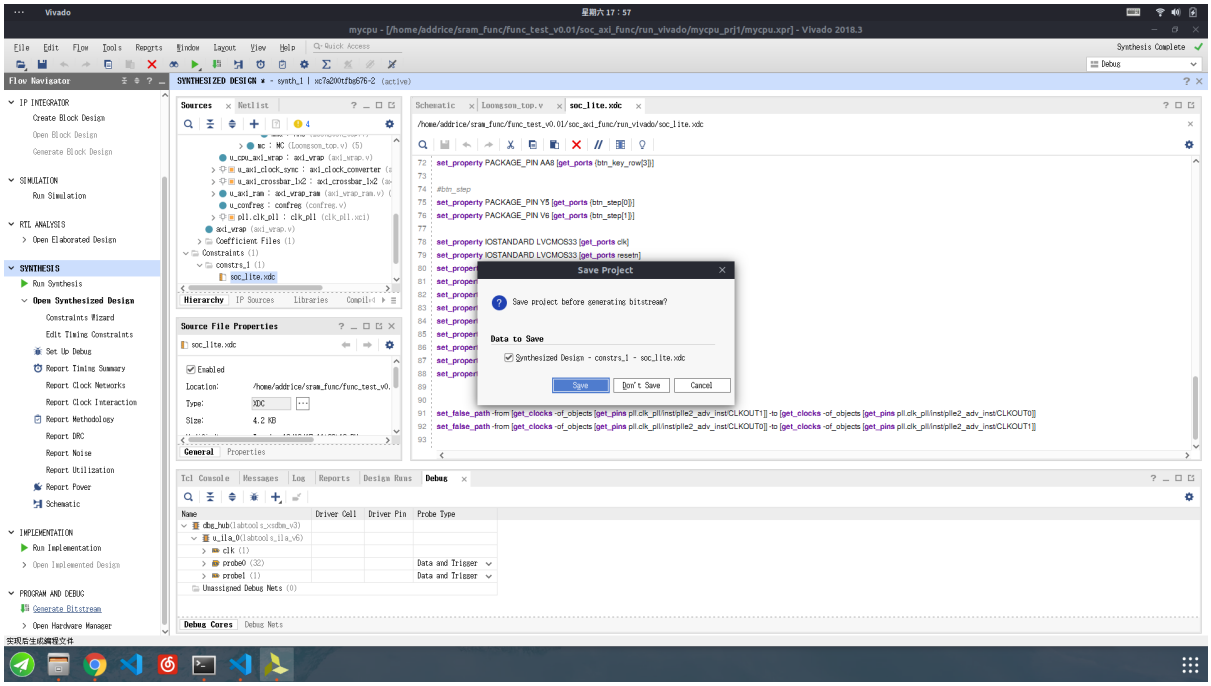
然后综合 (Run synthesis), 完成后 open Synthesized Design, 点击 Set up Debug，如图所示



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



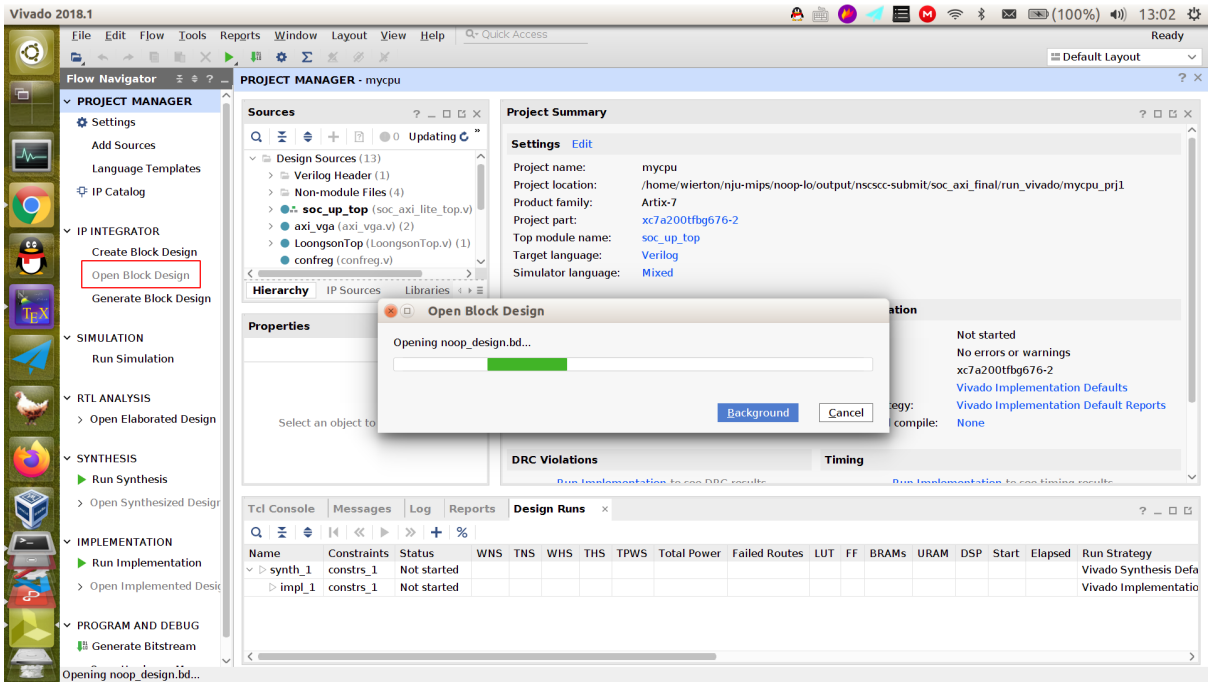
然后一路 next 完成。这一操作会修改我们的引脚文件，点击生成 bitstream，save 我们对综合文件的修改，如图所示



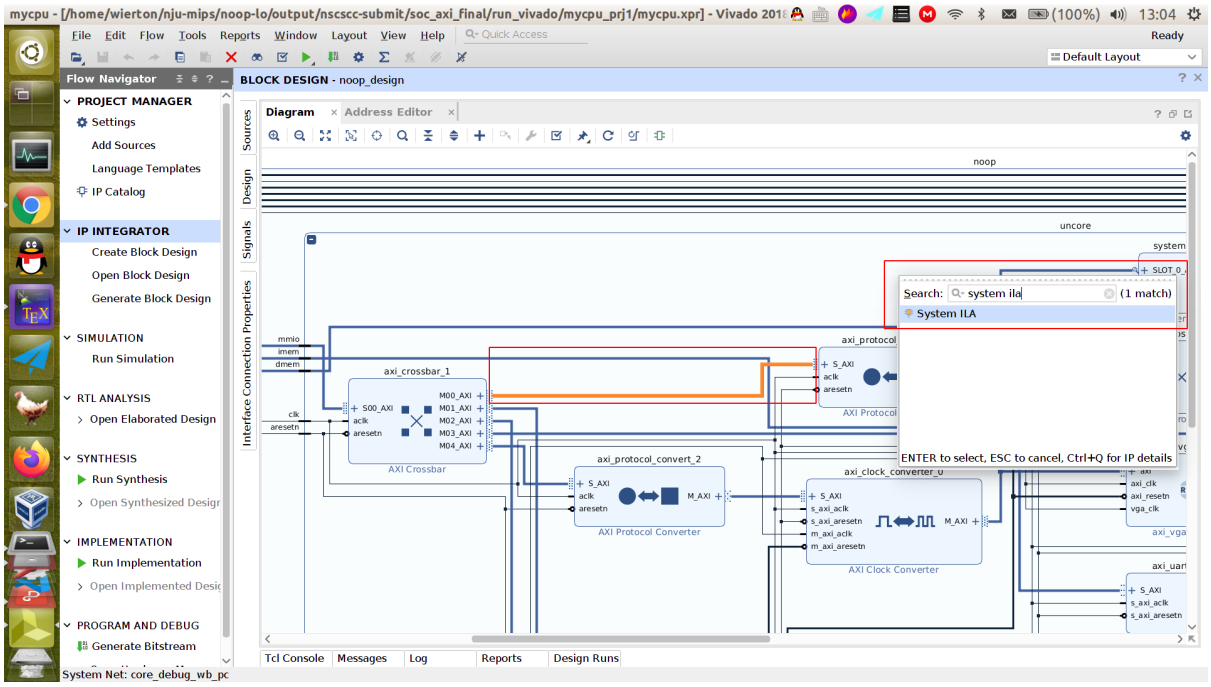
等待 bitstream 生成完毕上板即可。

1.6.2 block design 上使用 system ila

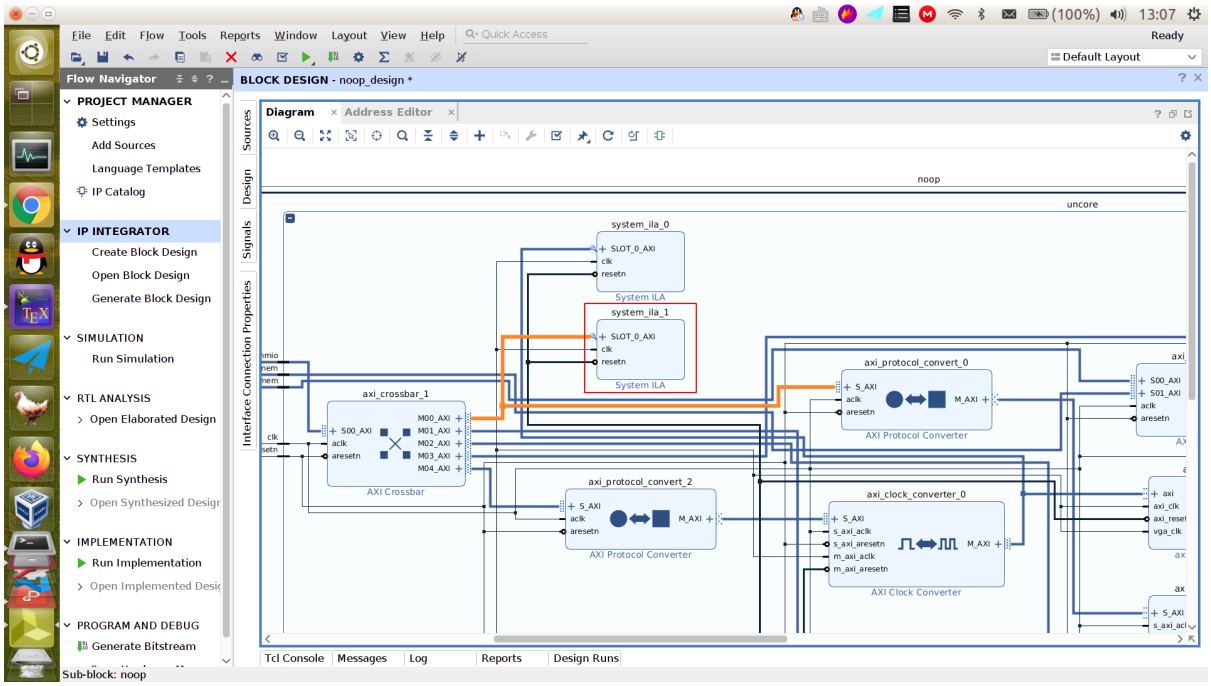
system ila 用来抓取 AXI4 的信号，相较于用(*mark_debug = "true"*)，system ila 抓取的信号易读性更强，并且可以直接在 block design 中操作。如下图所示，首先打开 block design：



然后在 block design 的面板上右键 add IP，搜索 system ila：



最后将 system ila 的 clock、reset 和所需要采样的信号连好：



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 minilab 讲义

minilab 的目的旨在让你能够快速掌握大实验所需的各种工具链，整体还是服务于大实验的。

2.1 Kconfig

开始实验前请先下载代码：`git clone git@github.com:nju-mips/minilab`，clone 下来之后你的文件结构是如下这样：

```
minilab
+-- kconfig
+-- M1
```

其中 kconfig 是 linux 内核所使用的自动化配置工具，也是这个小实验的重心，不过你并没有必要去研究它的源代码，我们所需要做的是学会去使用它。

下载下来源码之后，进入 M1 目录，尝试 make 一下，你会得到如下输出：

```
../kconfig/conf ./Kconfig
*
* This is a simple Kconfig for lab1
*
Algorithm selection
> 1. bubble sort (BUBBLE_SORT) (NEW)
  2. selection sort (SELECTION_SORT) (NEW)
choice[1-2?]:
use printer to print the array (USE_PRINTER) [N/y/?] (NEW)
#
# configuration written to .config
#
../kconfig/conf --syncconfig ./Kconfig
make: *** No rule to make target 'build/main.o', needed by 'build/M1'.  Stop.
```

重点是最后一行，`No rule to make target 'build/main.o'`，你需要做的第一件事便是打开 M1/Makefile.compile，并在里面添加.o 文件的规则。为求方便的你可能会将 Makefile.compile 修改成：

```
build/main.o: src/main.c
  mkdir -p build
  $(CC) -c $(CFLAGS) src/main.c -o build/main.o
```

但是这样一来假设项目中有几十个.c 文件你便需要写上几十条这样的规则，这种做法是低效且易错的。所幸为了消除这种重复性，Makefile 提供了`%`语法，它出现在规则中有通配符的含义，你可以用它来只写出一条规则便能自动编译所有的.c。

等你搞定了每个.o 的编译规则之后，再次键入`make`，你会得到如下的输出：

```
/tmp/ccJtMIec.o: In function ‘main’:
main.c:(.text.startup+0x7): undefined reference to ‘algo’
main.c:(.text.startup+0x2c): undefined reference to ‘algo’
main.c:(.text.startup+0x33): undefined reference to ‘printer’
main.c:(.text.startup+0x58): undefined reference to ‘printer’
collect2: error: ld returned 1 exit status
Makefile.compile:3: recipe for target 'build/main.o' failed
make: *** [build/main.o] Error 1
```

这个错误的含义很明显，链接的时候找不到`algo`这个符号，但当你打开 bubble_sort.c 的时候，你却能发现`algo`赫然躺在文件的末尾。这是为什么，明明定义了这个符号，却还是无法链接?

为了回答这个问题，我们需要引入条件编译这个概念。不过不是指`#if xx`这种源代码级别的条件编译，这里的条件编译是指在项目构建的时候，自动根据配置文件来决定需要编译链接哪些文件，这在 linux/u-boot/coreutils/busybox 中均是常用的技巧。在这里也是类似的，你需要阅读 Makefile 和 src/*.c，并在 Makefile 中添加适当的代码来让项目可以编译 (注意到这里为止不可以修改 src 目录下的文件)。同时为这个实验我们也设计了相应的测试脚本 testall.sh，如果你正确的实现了条件编译，则应该可以通过前六个测试，如下所示：

```
PASS bubble-sort
PASS selection-sort
PASS bubble-sort normal-printer
PASS selection-sort normal-printer
PASS bubble-sort reverse-printer
PASS selection-sort reverse-printer
FAIL reverse-sort normal-printer
FAIL reverse_sort_defconfig
```

在搞定了基本的编译之后，你可以尝试执行一下`make menuconfig`，并在这之后观察一下 M1 目录下有什么变化，M1/include 下面又有什么变化，这些变化会对这个项目产生什么影响? 如果不进行`make menuconfig`直接编译，整个项目会发生什么？你可以尝试修改一下 Kconfig 文件，然后再`make menuconfig`，看看有哪些变化。

经历了上述过程，你可能已经理解最后两个测试是要干什么的了。虽然如此，我们还是需要按部就班一下。我们先来讲解一下 src 和 include 下的几个源码文件干了一件什么事。在 common.h 头文件中定义了`struct algo`和`struct printer`，前者是给一个数组排序，后者是打印一个数组。在 main.c 里面会声明一个 algo，然后用它来进行排序。在 bubble_sort.c 和 selection_sort.c 里面各自都定义了`algo`这个变量，显然如果同时链接这两个文件是会链接失败的。

第七个测试需要你自行添加一个新的排序算法`reverse-sort`，它的功能是将输入按从大到小排好序。同时你需要修改 Kconfig，添加相应的选项`REVERSE_SORT`，并在 Makefile 里面添加相应的条件编译规则，使得你的修改能够通过第七个测试。

如果你接触过内核的编译，可能会用过`make xxx_defconfig`这样的功能。初见时可能感到惊奇，但仔细想一下你可能就知道这个功能是怎么实现的了。第八个测试也是为了测这个功

能，具体来说你需要添加一个`%_defconfig`的伪目标，这个伪目标会在被执行时去 configs 目录下将相应的配置文件覆盖到项目当前的配置文件上去。当然配置文件需要你自己准备，具体准备什么样的配置文件可以通过测试你可以通过阅读测试脚本来获得。

2.2 dts 和配置运行 linux

3 MIPS 相关

3.1 nju-mips 简介

3.1.1 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
- 乱序双发射 CPU

3.1.2 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 (该单元包括 CP0 和 TLB)，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

3.1.3 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

3.1.4 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

3.2 关于 PRU 的一些实现细节

PRU 全称 PRiviledged Unit，用来实现 CP0 和 TLB。

3.2.1 必要的 cache 指令

mips 不同于 x86，ICache 与 DCache 之间的一致性需要通过 cache 指令来维护。但是具体实现的时候无需把 cache 指令的所有功能都实现一遍，cache 指令的某些操作码并没有被使用过。表3记录了 linux 和 u-boot 中所使用的 cache 指令。

cache 指令	操作码	作用的 cache	使用范围	具体功能
`cache 0x00, [addr]`	`5b000'00`	指令 cache	linux	
`cache 0x08, [addr]`	`5b010'00`	指令 cache	linux	
`cache 0x10, [addr]`	`5b100'00`	指令 cache	linux+u-boot	
`cache 0x14, [addr]`	`5b101'00`	指令 cache	linux	
`cache 0x01, [addr]`	`5b000'01`	数据 cache	linux	
`cache 0x09, [addr]`	`5b010'01`	数据 cache	linux	
`cache 0x15, [addr]`	`5b101'01`	数据 cache	linux+u-boot	
`cache 0x03, [addr]`	`5b000'11`	L2 cache	linux	
`cache 0x07, [addr]`	`5b001'11`	L2 cache	linux	
`cache 0x13, [addr]`	`5b100'11`	L2 cache	linux	
`cache 0x17, [addr]`	`5b101'11`	L2 cache	linux	
`cache 0x0b, [addr]`	`5b010'11`	L2 cache	linux	

表 3: 常用 cache 指令

3.2.2 异常实现的注意点

关于异常，最需要注意的一点是它和异常码不是一一对应的，异常跳转入口可以视为是由异常和异常码共同决定的：

异常	会产生的异常码
TLB Refill Exception	TLBL, TLBS
TLB Invalid Exception	TLBL, TLBS
TLB Modified Exception	TLBMod

表 4: 需要注意的异常和异常码

下述为处理异常发生时如何跳转的代码，原版在 nemu-mips32/src/cpu/cpu.c 中：

```
void launch_exception(int exception, int excode) {
  // 注意，异常和异常码不是一一对应的
  uint32_t vecOff = 0;
```

```
  if (cpu.cp0.status.EXL == 0) {
    if (cpu.last_instr_in_delayslot) {
      cpu.cp0.epc = cpu.pc - 4; // 跳转指令的地址
      cpu.cp0.cause.BD = 1;
    } else {
      cpu.cp0.epc = cpu.pc;     // 最后执行的指令的地址
      cpu.cp0.cause.BD = 0;
    }

    // 这里计算异常入口地址的偏移量
    if (exception == EX_TLB_REFILL)
      vecOff = 0x000;
    else if (excode == EXC_INTR && cpu.cp0.cause.IV == 1)
      vecOff = 0x200;
    else
      vecOff = 0x180;
  } else {
    // 注意这里不设置epc
    vecOff = 0x180;
  }

  cpu.cp0.cause.ExcCode = code;
  cpu.cp0.status.EXL = 1;
  // 注意这里，异常跳转地址的计算方式是[基址+偏移量]
  if (cpu.cp0.status.BEV == 1) {
    cpu.br_target = 0xbfc00200 + vecOff;
  } else {
    cpu.br_target = 0x80000000 + vecOff;
  }
}
```

3.2.3 tlb 实现细节

关于查询 tlb 的过程中如何发起异常，可以参见 nemu-mips32/src/cpu/mmu.c，下述代码对原版做了简化。

```
void tlb_exception(int ex, int code, vaddr_t vaddr, unsigned asid) {
  cpu.cp0.badvaddr = vaddr;
  cpu.cp0.context.BadVPN2 = vaddr >> 13;
  cpu.cp0.entry_hi.vpn = vaddr >> 13;
  cpu.cp0.entry_hi.asid = asid;
  launch_exception(ex, code);
}

vaddr_t page_translate(vaddr_t vaddr, int rwbit) {
  uint32_t excode = rwbit == MMU_LOAD ? EXC_TLBL : EXC_TLBS;
  uint32_t va_31_13 = (vaddr & ~0x1FFF) >> 13;
```

```
  for (int i = 0; i < NR_TLB_ENTRY; i++) {
    uint32_t mask = tlb[i].pagemask;
    assert(mask == 0 || (mask | (mask + 1)) == 0);
    bool match = (tlb[i].vpn & ~mask) == (va_31_13 & ~mask) &&
                 (tlb[i].g || tlb[i].asid == cpu.cp0.entry_hi.asid);
    if (!match) continue;

    bool EvenOddBit = vaddr & ((mask + 1) << 12);
    /* match the vpn and asid */
    tlb_phyn_t *phyn = EvenOddBit ? &(tlb[i].p1) : &(tlb[i].p0);
    if (phyn->v == 0) {
      tlb_exception(EX_TLB_INVALID, excode, vaddr, tlb[i].asid);
      return -1u;
    } else if (phyn->d == 0 && rwbit == MMU_STORE) {
      tlb_exception(EX_TLB_MODIFIED, EXC_TLBM, vaddr, tlb[i].asid);
      return -1u;
    } else {
      // # pfn_PABITS-1-12..0 corresponds to pa_PABITS-1..12
      //
      // pa   pfn_PABITS-1-12..EvenOddBit-12 || va_EvenOddBit-1..0
      // case TLB[i].Mask
      //  2#0000000000000000: EvenOddBit   12
      //  2#0000000000000011: EvenOddBit   14
      //  2#0000000000001111: EvenOddBit   16
      // endcase

      uint32_t highbits = (phyn->pfn & ~mask) << 12;
      uint32_t lowbits = vaddr & (((mask + 1) << 12) - 1);
      return highbits | lowbits;
    }
  }

  tlb_exception(EX_TLB_REFILL, excode, vaddr, cpu.cp0.entry_hi.asid);
  return -1u;
}
```

3.3 外设地址段

地址段	外设
0x80000000 - 0x8fffffff	DDR (cached)
0xa0000000 - 0xafffffff	DDR (uncached)
0xb0000000 - 0xb0000fff	nemu GPIO trap
0xb0002000 - 0xb0002fff	nemu rtc
0xb0003000 - 0xb0003fff	nemu screen config
0xb0400000 - 0xb04fffff	nemu video memory
0xbfc00000 - 0xbfcfffff	SRAM(用于存放引导程序)
0xbfe50000 - 0xbfe50fff	xilinx uartlite serial
0xbfe80000 - 0xbfe80fff	xilinx spi flash
0xbfe94000 - 0xbfe94fff	nemu Keyboard
0xbfe95000 - 0xbfe95fff	npc PerfCounter
0xbfe96000 - 0xbfe95fff	PS/2 keyboard
0xbff00000 - 0xbff0ffff	xilinx emaclite

3.4 CLZ 的递归高效实现

3.4.1 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)
  }
}
```

3.4.2 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)
}
```

4 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。

4.1 测试相关

4.1.1 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。

4.1.2 tlbtest

这个是从龙芯的测试集中扒出来的，用于进行 tlb 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。

4.1.3 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

4.1.4 nexus-am/apps/coremark

基于 AM 的评分程序，workload 中等。

4.1.5 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。

4.2 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，如果想了解某个外设的 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

4.2.1 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
```

4.2.2 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。

4.2.3 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。

4.3 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。

5 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。

5.1 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@bff00000 using MAC address from ROM
eth0: ethernet@bff00000
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] 1fe50000.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] 0x000000000000-0x000000800000 : "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：

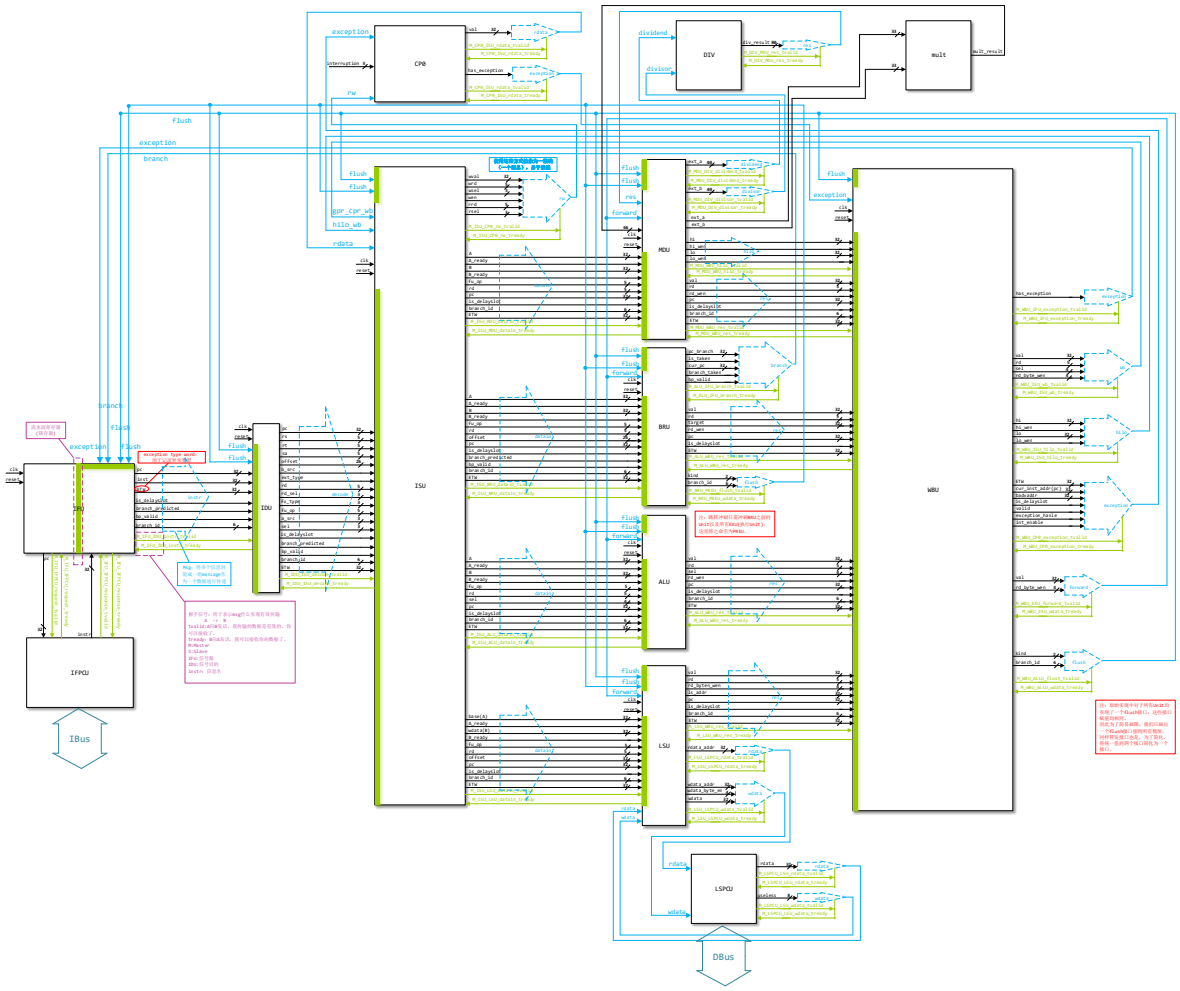
```
nemu: src/cpu/cpu.c:221: vaddr_write: Assertion 'dev && dev->write' failed
CPUAssert message: bad addr bfe80060

[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。

6　woop 设计

7　往届 core 设计 (2017)

详细内容可以查看原文档。



8 往届 core 设计 (2018)

8.1 模块设计图纸

我们的 cpu 整体设计为 6 段流水线，如图1所示。由于初赛数据集较小，大部分数据集都可以直接装在 l1 cache，l2 cache 的设置在初赛中反而在某些数据集上起反效果，所以在最终提交的源码中，我们移除了这一组件。以及图中并未显式画出 flush 信号，这一信号用于分支跳转以及中断异常的冲刷。

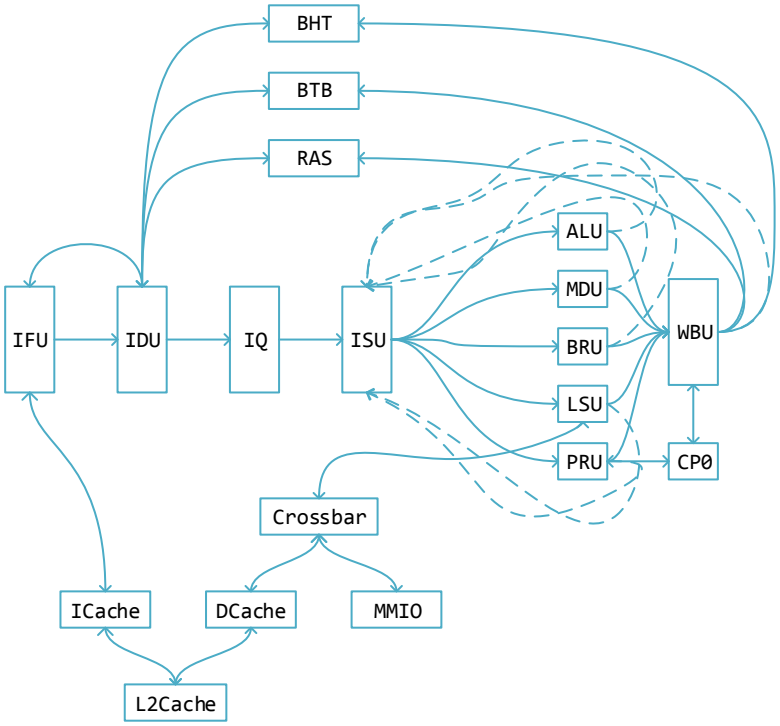


图 1: CPU 的核内设计

图中所有的执行单元和写回单元 (wbu) 均有一条虚线连接到 ISU，这条虚线的含义是 bypass 信号，由于实际数据写回至少会晚上 3 个周期，为了能够更快的获取到寄存器数据，防止 ISU 因为等待数据就绪而空耗时间，项目添加了如图中虚线所示的 bypass 信号。

除了 core，为了能够对接到龙芯的开发板，我们加入 block design 用于处理外围引线，其中 block design 的设计如图2所示：

8.2 core 模块简介

CPU 核心的架构如下图所示：

首先是 IFU，取指单元。负责从 icache 中将指令取上来。由于 icache 存储使用的是 block ram，而 block ram 是在下一周期返回读结果的，所以相应的，IFU 是两级流水。第一级流水将 PC 发给 icache，第二个周期从 icache 中将数据取上来。

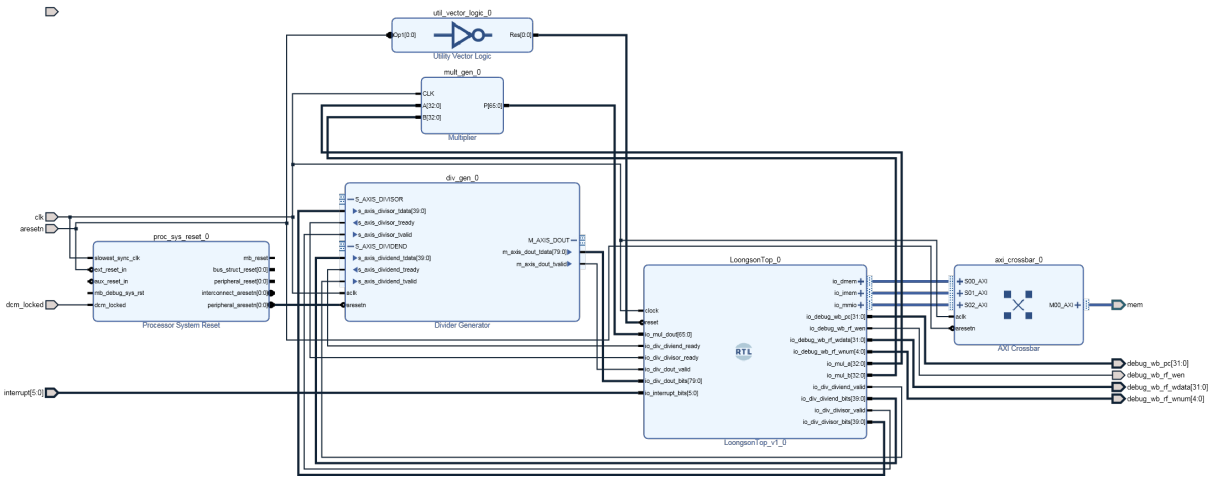


图 2: CPU 的核外布线

接下来是 IDU，译码模块。由于我们实现的是初步的乱序结构，在译码阶段，除了要完成指令的译码外，还要完成乱序需要的功能：寄存器重命名：对于源寄存器，要读取 scoreboard，读取 rename table 对于目的寄存器需要分配物理寄存器（处理 WAW）分配 ROB 分配 Branch ID 分配 Load Store queue 项，

分支预测：分支预测模块包含了三样东西：BHT，branch history table，负责进行分支预测，我们实现的是类似 gshare 的分支预测器。BTB，branch target table，负责预测间接跳转指令的跳转地址。RAS，RAS 负责预测函数返回的跳转地址。

计分板 scoreboard 中主要有：32 项的 status，记录每个寄存器现在的状态。寄存器共有三种状态，他们的含义分别是：SB_READY：就绪态，这个寄存器的最新结果就在寄存器中。SB_BUSY：寄存器已经被重命名到 ROB 了，但是结果尚未就绪。SB_ROB_READY：此寄存器已经被重命名到 ROB 中了，结果就绪，但是此指令尚未 retire，所以结果要从 ROB 中读取。

ISU 然后是 ISU，即指令发射队列，译码后的指令进入发射队列中，当源操作就绪，并且后端执行单元空闲时，即可读取发射操作数，并发射到后端执行单元执行。

后端执行单元：后端执行单元执行完指令后，将指令状态以及计算结果写入 ROB 中，并通过 bypass network 传递到 ISU。

ALU：计算算术逻辑指令 MDU：计算乘除法指令 PRU：处理 cp0 指令 BRU: 处理分支跳转指令

LSU: 处理 load store 指令 LSU 以及 Load Store queue 的设计 LSU 主要的工作是：进行译码，地址生成，以及地址转换（查 TLB），将指令派发到 Load Queue 以及 Store queue。如果这条指令是 load，则派发到 Load queue，如果是 Store，则派发到 store queue。load queue、store queue 的释放都是在指令 retire 后进行的。

ROB reorder buffer，存储指令计算的结果，以及指令的例如 pc、异常等基本信息。ROB 是一条有序队列，每个执行单元到 ROB 都有一个写口，另外 ROB 有两个读口，用于 ISU 从中读取数据，另外 ROB 还有一个 commit 口。

WBU 当 ROB 头部的指令就绪时，它就可以 commit 了。然后它就进入了 WBU，写回单元。写回单元检查指令是否出了异常，并进行相应的处理。另外，对于 non-speculative load/store，

WBU 将他们对应的项从 Load store queue 中唤醒，并执行。

TLB 我们实现了 64 项 TLB，支持虚拟内存。TLB 本质上是一个大的 CAM 查找表。

L1 ICache、DCache L1 ICache 负责处理来自 IFU 的请求，DCache 处理来自 load store queue 的请求，L1 ICache 与 DCache 均用 block ram 实现，均为两级流水，在 cache hit 时，能一直流水。在 cache miss 时，则阻塞。另外 L1 DCache 支持背靠背的 load，store，连续的 load、store，store、load 之间的数据转发均可以处理。

L2 cache 我们实现了 L2 cache，AXI4 接口，blocking，一次只能只能处理一个请求。由于龙芯测试集 working set 很小，所以在性能测试中，我们没有使用这个 L2 cache。

8.3 实现细节

8.3.1 未重命名的指令以及寄存器

此外由于我们现在暂时没有对 cp0 寄存器以及 Hi，Lo 寄存器进行重命名，为了保持依赖，同时为了实现简便。现在在 IDU 阶段，cp0 指令以及乘除法指令或者对上述寄存器的所有操作，都必须要等 rob 空了才会接着执行。IDU 负责识别这些指令，并对前端进行必要的阻塞。

8.3.2 发射队列的实现

然后是 ISU，即指令发射队列，译码后的指令进入发射队列中，发射队列中的每一项在进来的时候，都会记住自己的源寄存器的状态（在 IDU 阶段从 scoreboard 中读取），以及被重名到了 ROB 的哪一项。如果指令的源操作数的状态是 SB_READY 或者 SB_ROB_READY，则可以被直接发射，并不需要等待依赖满足。但是如果某一个操作数的状态是 SB_BUSY，则这条指令需要待在 Issue queue 中，等待唤醒。

issue queue 中的指令主要要满足的依赖有：源操作数就绪，执行单元空闲。

8.3.3 唤醒操作

我们从后端的每一个执行单元，以及 WBU 写回单元，都拉出一组线到 issue queue，包含要结果就绪的寄存器下标、rob 项下标以及算出来的值。我们将寄存器以及 rob 地址，与 issue queue 中的每一项进行比对，判断该条指令是否有源操作数就绪。这构成了 wake up 操作。另外，传递过来的算出来的新值，构成了 bypass network。

8.3.4 发射

源数据就绪的指令，即可被发射，由于受发射宽度的限制（现在为 1），我们用 priority encoder 选择一条指令出来发射。指令在发射时读取源操作数，源操作数来自于：寄存器，ROB 或者 bypass network。Speculative load、store。由于对 mmio 的 load/store 以及对 memory 的 store 是有副作用的，为了实现精确内存，这些 load store 是必须在指令 retire 时才能执行的。而对 memory 的 load，是可以推测执行的。

8.3.5 精确内存，speculative load store

由于 mmio load/store，memory store 是有副作用的，仅 memory load 无副作用。所以 load queue 中对 memory 的 load 是可以推测执行的，而对 load queue 中对 mmio 的访问，以及 store queue 中的任务是在指令 retire 后执行的。

8.3.6 dynamic memory disambiguation 以及维护 memory RAW dependency

由于指令是乱序发射并且 load 是允许推测执行的，所以可能出现新的 load 在老的 store 之前被 issue 到 lsq，被执行完成，造成依赖关系错误。由于我们将 load queue 以及 store queue 实现成了两条无序 queue，无法单从 queue 的位置关系，判断 load，store 指令间的先后关系。同时，我们也没有实现 store queue 到 load queue 之间的 bypass。所以我们是在 store 指令 retire 时，将其地址与 load queue 中的每一项进行匹配，如果有匹配上的，并且已经执行的 load，则说明出现了 dependency failure，然后通过冲刷流水线来解决。这样子做是可行的，因为 store 指令是在 retire 时执行，此时，我们可以保证 load queue 和 store queue 中的指令都比它年轻。

8.3.7 ICache、DCache 一致性

我们实现了 Cache 指令，支持 ICache、DCache 一致性。

8.4 乱序处理器

8.4.1 从 I4 到 IO2I

我们在初赛时提交的版本是一个全顺序的流水线，In order fetch，in order issue, in order execute, in order retire，简称 I4。在初赛后，我们对其进行了下面这些提升，将性能测试跑分提升到了 52 分：

1. 更低的乘除法器延迟
2. 原有的实现中的 cache line 的 valid，dirty 均是用触发器实现，因为无法综合成 BRAM，占据了大量的资源，影响了频率。在解决这个问题后，频率从 70M 提升到 90M。
3. 提升分支预测准确率：
   (a) perf test 中的绝大部分测试需要很短的 history，小部分，如 bubble sort、quick sort 需要很长的 history，因此我们使用了两个 history 长度不同的 BHT，并取预测效果最好的作为分支预测结果。
   (b) RAS speculative update。由于流水线较长（主要是由于中间 instruction queue 的存在），而我们原来 RAS 压栈出栈都是在 WBU 阶段进行的，导致对于非常小的函数调用，call 尚未 commit，并将返回地址压入 RAS，return 就已经进了流水线了。我们支持了让 RAS 支持 speculative update，在 IDU 阶段遇到 call，return，就更新 RAS。由于是 speculative update，所以当出现分支预测错误时，RAS 会进行部分恢复（恢复 stack top 指针）。添加了这个后，对于小函数调用多的 stringsearch，提升明显，几乎所有的 test 的 return 预测错误，都降到了接近 0

4. DCache store、load bypass。对于 stringsearch、stream copy 等测试，存在明显的 store，load 背靠背的情况，我们处理了背靠背地址冲突的情况，加上了 bypass。明显提升了这些项的分数。

8.4.2 IO2I：初步的乱序处理器

in order fetch，OoO issue，OoO execute，In order retire。此款处理器能够运行在 70M，跑分 33 分左右。

考虑到乱序的复杂性，为了保证正确性，尽早拿出一个能够运行的版本，我们在实现中做了如下取舍，这保证了实现的低复杂度与正确定，但是性能不太好，甚至不如 I4 版本处理器。

1、显式重命名（重命名到物理寄存器）VS 隐式重命名（重命名到 ROB）重命名主要要处理好：

scoreboard：寄存器 ready，busy 状态物理寄存器资源分配

对于 scoreboard，这两种方案复杂性差不多。但是对于物理寄存器资源的分配，差别则很大。显式重命名需要实现 active list，free list 来维护物理寄存器的分配，在遇到分支时，需要做 active list 以及 rename table 进行 check point，分支预测错误时需要进行回滚，复杂度更高。同时在异常时，需要 walk back rob，复杂度太高。上述这些可能看起来还不是太复杂，但是要考虑到：在 checkpoint 以及恢复 free list，scoreboard 以及 rename table 时，这些资源的分配，回收在同时进行，corner case 非常多，不容易写对。一不小心就容易出现：1、资源泄露，程序卡死（free list 没有回收好）2、依赖处理错误，程序错误（scoreboard 没有处理好）3、资源被重分配，程序错误（rename talbe 没有处理好）

而隐式重命名只需要维护 ROB，flush 直接 flush rob 即可，不需要复杂的 checkpoint 以及恢复操作。

2、分支预测错误冲刷流水线的时机问题：由于 ROB 的存在，如果在 retire 时冲刷，会导致分支预测错代价太大了，分支预测错要尽早 flush。按理说，对于重命名到 ROB 的实现，只要将 branch 后的指令全部从 rob 中刷掉就可以了，由于我们需要 scoreboard 来维护寄存器是否 ready，所以 scoreboard 必须要做 checkpoint，这就又是一个复杂的点。另外，由于我们要支持延迟槽指令，即使 branch 是放到了 retire 时冲刷，我们也不能直接对 scoreboard 进行 reset，因为此时延迟槽指令还在流水线中。所以对于这种方案，我们也要对 scoreboard 进行 checkpoint，并在 flush 时恢复。为了让实现简单正确，我们采取的是分支预测错的 flush，要等到延迟槽指令 retire 时，才发出，这样子可以保证流水线中没有任何处理器，scoreboard 也可以直接 flush，而不必进行 checkpoint。

3、load store queue 的实现对于 load store queue，由于我们采取的是 load queue，store queue 都是无序 queue 的实现，load 指令无法判断哪条 store 比它老，因此也就无法进行数据 bypass。我们采取了更加简单的方式来保证 RAW dependency，store 时，如果发现了有比它新的 load 被执行了，那就 flush 流水线。

4、简单实现的资源占用问题对于单发射的乱序处理器，理论上，wake port 可以只有一个，ROB 的读口应该可以控制到三个（两个用来读数据，一个用来 commit），写口应该限制到一个。我们为了简单起见，所有的 FU 都是直接 wake up issue queue，都直接将结果写入 ROB，这导致

了 ROB 的写口数量暴增，issue 的 CAM 逻辑资源使用量也很大，进而限制了主频，以及 issue queue 与 ROB 的项数，进而限制了性能。

5、对 Hi，Lo 没有重命名，乘除法指令等待 ROB 空了再发射，这严重影响了 coremark 等乘除法密集型应用的性能。

如图1所示，图中列出了主要的 core 设计所涉及的模块，其中各个模块的含义大致描述如下：

1. IFU 为取指单元，其通过握手信号向 ICache 发送取指信号，为了防止照成关键路径，ICache 内部数据存储使用的是 block ram，所以 ICache 在收到数据的下一周期才能返回数据，同时 ICache 的内部实现了流水化，所以迟一个周期返回数据几乎没有影响。
2. IDU 为译码单元，用于对 IFU 取得的指令进行译码，以取得这条指令的操作数寻找模式，操作数扩展方式，访问的寄存器编号，写回的寄存器编号，所需发往的执行单元编号。
3. IQ 为指令队列，用于存储所有已经译码成功的指令，为了防止后端执行周期过长阻塞前端。
4. ISU 为分发单元，通过译码所得的结果读取寄存器，同时在寄存器读写约束不满足的时候进行必要的等待，索引转发结果。
5. ALU 为算术运算单元，用于完成加法、减法、移位等可在一个周期内出结果的运算指令
6. MDU 为乘除法单元，用于完成乘法和除法的运算，这一模块主体执行功能放在了核外的 IP 核上，而 MDU 内部逻辑主要是等待乘除法器的执行结果。
7. BRU 为分支跳转单元，通过对 ISU 传入的操作数进行计算，以获取跳转目标，同时对于条件跳转指令还需要计算一下是否跳转
8. LSU 为访存单元，用于 load store 指令的执行，若 dcache miss，还需要额外的周期等待访存结果
9. PRU 为特权指令单元，非特权指令不允许发送到这个单元，需要发送到这个单元的指令有 MFC0、MTC0、SYSCALL、ERET，同时对于取指异常的指令和译码异常的指令，也需要发送到这个单元用以下拉异常信号
10. CP0 为 0 号协处理器，用于完成中断异常主要计算功能
11. WBU 为写回单元，用于将执行单元获取到的操作数写回到目标寄存器
12. BHT、BTB、RAS 为分支预测单元，由于延迟槽的存在，整体的分支预测请求由 IDU 发出，并用于重定位 IFU，wbu 在写回寄存器时，也需要将 BRU 的计算结果更新到分支预测单元

9 往届 core 设计 (woop)

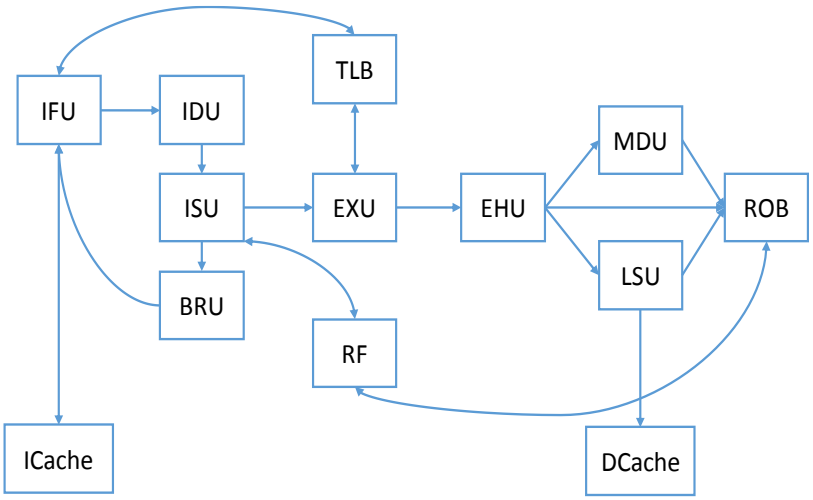


图 3: CPU 的核内设计