AM编译过程

准备工作

准备交叉编译环境

Shell | 复制代码

```
1  sudo apt-get install g++-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

获取源代码

```
git clone https://github.com/NJU-ProjectN/am-kernels.git
git clone -b ysyx2021 https://github.com/NJU-ProjectN/abstract-machine.git
```

设置环境变量

方法一：直接在当前目录下使用

```
export AM_HOME=$(pwd)/abstract-machine
```

方法二：将AM_HOME写入.bashrc中（若使用其它shell，请手动将环境变量添加到相应的rc文件中）

```
echo export AM_HOME=$(pwd)/abstract-machine >> ~/.bashrc
source ~/.bashrc
```

推荐采用第二种方法，这样在重新打开shell时会自动完成AM_HOME的配置。

运行如下命令

```
echo $AM_HOME
cd $AM_HOME
```

可以看到echo命令正确输出了AM项目的路径，而cd命令则正确地进入了目标目录。若echo命令输出不正确，请检查上述环境变量的配置。

abstract-machine和am-kernels

AM

AM，也就是目录下的abstract-machine项目，为程序提供了裸机运行时环境，它的文件组织如下所示

Git 复制代码

```
abstract-machine/
├── am
│   ├── include
│   │   ├── amdev.h
│   │   ├── am.h
│   │   └── arch
│   ├── Makefile
│   └── src
│       ├── mips32.h
│       ├── mycpu
│       │   ├── cte.c        # CTE: 上下文扩展, 为程序提供上下文管理的能力
│       │   ├── input.c
│       │   ├── ioe.c        # IOE: 输入输出扩展, 为程序提供输出输入的能力
│       │   ├── libgcc
│       │   ├── mpe.c        # MPE: 多处理器扩展, 为程序提供多处理器通信的能力
│       │   ├── start.S      # 程序入口
│       │   ├── timer.c
│       │   ├── trap.S
│       │   ├── trm.c        # TRM: 最简单的运行时环境, 为程序提供基本的计算能力
│       │   └── vme.c        # VME: 虚存扩展, 为程序提供虚存管理的能力
│       ├── native
│       ├── nemu             # 以NEMU为平台的AM实现
│       ├── riscv32.h
│       ├── riscv64.h
│       ├── x86
│       └── x86.h
├── klib                     # 常用函数库
│   ├── include
│   ├── Makefile
│   └── src
│       ├── cpp.c
│       ├── int64.c
│       ├── stdio.c
│       ├── stdlib.c
│       └── string.c
├── Makefile
└── scripts                   # 构建/运行二进制文件的makefile
    ├── isa
    ├── mips32-nemu.mk
    ├── native.mk
    ├── platform
    ├── riscv32-nemu.mk
    ├── riscv64-mycpu.mk      # 在一生一芯项目中只使用本文件
    ├── riscv64-nemu.mk
    ├── x86_64-qemu.mk
    ├── x86-nemu.mk
    └── x86-qemu.mk
```

目前文件主要分为两大部分

- abstract-machine/am: 不同架构AM api的实现，目前我们只需要关注mycpu的相关内容
- abstract-machine/klib: 架构无关的库函数，方便应用程序开发

当然，里面的绝大部分函数都没有实现，如果应用程序调用了这些函数，就需要你把它们实现了。

am-kernels

am-kernels收录了一些可以在AM上运行的测试集和简单程序

```Shell
am-kernels
├── benchmarks                  # 可用于衡量性能的基准测试程序
│   ├── coremark
│   ├── dhrystone
│   └── microbench
├── kernels                     # 可展示的应用程序
│   ├── hello
│   ├── litenes                 # 简单的NES模拟器
│   ├── slider                  # 简易图片浏览器
│   ├── thread-os               # 内核线程操作系统
│   └── typing-game             # 打字小游戏
└── tests                       # 一些具有针对性的测试集
    ├── am-tests                # 针对AM API实现的测试集
    └── cpu-tests               # 针对CPU指令实现的测试集
```

我们先关注cpu-tests中的程序，比如其中的add.c程序，它不断计算每组数的和是否等于预期结果，如果不等于预期结果，那么说明我们cpu指令实现有误，它通过halt(1)退出程序，如果通过了所有检查，那么main函数正常退出，返回0。

```c
void check(bool cond) {
  if (!cond) halt(1);
}

int test_data[] = {0, 1, 2, 0x7fffffff, 0x80000000, 0x80000001, 0xfffffffe
...};
int ans[] = {0, 0x1, 0x2, 0x7fffffff, 0x80000000, 0x80000001, 0xfffffffe
...};

#define NR_DATA LENGTH(test_data)

int main() {
  int i, j, ans_idx = 0;
  for(i = 0; i < NR_DATA; i ++) {
    for(j = 0; j < NR_DATA; j ++) {
      check(add(test_data[i], test_data[j]) == ans[ans_idx ++]);
    }
    check(j == NR_DATA);
  }
  check(i == NR_DATA);
  return 0;
}
```

elf和bin文件的生成：从makefile说起

现在，我们希望将这个简单的add.c编译成.elf文件和.bin文件，只需要在cpu-test目录下执行

```shell
make ARCH=riscv64-mycpu ALL=add
```

就可以在build/目录下找到编译生成的文件了。

在输入这条命令后，make究竟在背后做了些什么呢？通过makefile可以看到，程序主要执行以下三条命令：

```git
@/bin/echo -e "NAME = $*\nSRCS = $<\nLIBS += klib\ninclude
$${AM_HOME}/Makefile" > $@
-@make -s -f $@ ARCH=$(ARCH) $(MAKECMDGOALS)
-@rm -f Makefile.$*
```

去掉命令前的@, 我们可以获取程序实际执行的命令

```Git
/bin/echo -e "NAME = add\nSRCS = tests/add.c\nLIBS += klib\ninclude ${AM_HOME}/Makefile" > Makefile.add
make -s -f Makefile.add ARCH=riscv64-mycpu
rm -f Makefile.add
```

可以看出，它的逻辑非常简单，第一条命令生成一个makefile，第二条命令执行它，最后删除生成的makefile。我们打开生成的makefile，发现它包含了另一个文件，继续打开这个文件，发现它比较复杂，不知道从哪里读起?

```Makefile
NAME = add
SRCS = tests/add.c
LIBS += klib
include /home/piper/program/abstract-machine/Makefile
```

那就让make自己告诉我们它做了些什么吧。我们同样去除am下makefile命令前的@，再次执行make,

```Shell
riscv64-linux-gnu-gcc -I[include path] -march=rv64ifd -mcmodel=medany -c -o add.o add.c

riscv64-linux-gnu-gcc -I[include path] -march=rv64ifd -mcmodel=medany -c -o mycpu/boot/start.o mycpu/boot/start.S
riscv64-linux-gnu-gcc -I[include path] -march=rv64ifd -mcmodel=medany -c -o mycpu/trm.o mycpu/trm.c
...

ar rcs am-riscv64-mycpu.a start.o trm.o timer.o ...

...
ar rcs klib-riscv64-mycpu.a string.o stdlib.o ...

riscv64-linux-gnu-ld -e _start -o add-riscv64-mycpu.elf add.o am-riscv64-mycpu.a klib-riscv64-mycpu.a

riscv64-linux-gnu-objdump -d add-riscv64-mycpu.elf > add-riscv64-mycpu.txt

riscv64-linux-gnu-objcopy -S -O binary add-riscv64-mycpu.elf add-riscv64-mycpu.bin
```

makefile首先将目标程序add.c编译成.o文件，之后分别编译将am和klib目录下的文件，并打包生成两个静态库am-riscv64-mycpu.a, klib-riscv64-mycpu.a，之后通过ld命令进行链接，生成elf文件，

再通过objcopy获取add程序的指令和数据，以二进制形式保存在.bin文件中。为了方便阅读程序包含的riscv指令，我们同时会使用objdump进行反汇编，打开build目录下的.txt文件就能看到add.bin包含的全部指令了。

AM/mycpu提供了什么

初始代码化和停机函数

我们现在回到AM，在ld命令中，我们通过-e _start 将程序的入口地址设置成了_start。_start在abstract-machine/am/src/mycpu/strat.S中定义，它会设置栈指针，并跳转到_trm_init函数执行

Makefile 复制代码

```
_start:
  mv s0, zero
  la sp, _stack_pointer
  jal _trm_init
```

在_trm_init中，调用add所定义的main函数，从而进入应用程序执行，这就到大家所熟悉的领域了。当main函数返回时，我们调用halt结束程序的运行。

Makefile 复制代码

```
void _trm_init() {
  int ret = main(mainargs);
  halt(ret);
}
```

在这里，我们的halt只是一个while(1)的循环，从反汇编结果也可以看出这条指令不断地跳转到它自身。

Makefile 复制代码

```
void halt(int code) {
  while (1);
}
```

为了让我们的cpu识别程序终止，可以在halt函数中插入一些内联汇编指令

Makefile 复制代码

```
void halt(int code) {
  asm volatile("mv a0, %0; .word 0x0000006b" : :"r"(code));
  while (1);
}
```

这几条内联汇编指令首先将main函数的返回值(退出码)放进a0寄存器中，再执行一条自定义指令。重新编译，现在可以在halt函数中找到我们插入的指令

```
Makefile | 复制代码
00050513            mv   a0,a0
0000006b            0x6b
```

我们选择riscv标准中保留的0x6b指令作为程序结束的标志，在cpu读取到这条指令时，它就能够知道应用程序已经执行完成，并且能够通过a0的值(也就是main函数返回值)判断程序运行状态。

前面已经介绍过，对于add程序，在它正常通过所有检查时，main函数最终返回0，而在它没有通过check检查时，会调用halt(1)，因此在cpu读取到0x6b指令时，只需要判断a0是否为1，就可以得知我们涉及的cpu是否通过了add测试了。

另外，0x6b只是我们自定义的一条结束指令，你也可以任意选择一条在rv中没有定义过的指令码作为你自己的结束指令。

putch

如果应用程序想要输出字符，比如说hello程序会不断调用putch打印字符。但我们还没介绍输入输出的内容，cpu也还没有实现输入输出的功能，不过我们可以参考上文介绍halt时添加自定义指令的方式，来在仿真环境中输出一个字符：当cpu执行到用于输出字符的自定义指令时，就可以通过$write()(Verilog中)或者printf()(Chisel中)输出寄存器中存放的字符。

```c
C | 复制代码
void putch(char ch) {
  asm volatile("mv a0, %0; .word ???" : :"r"(ch)); // 把???改成你选择的另一条自定义指令
}
```

更多扩展

在ppt中已经介绍过，AM = TRM + IOE + CTE + VME + MPE，我们这里只对TRM进行了初步的介绍，如果要运行更加复杂的应用程序，那么就要让我们的AM提供更加丰富的运行时环境，我们会在后续报告中介绍IOE和CTE。如果你对AM感兴趣，也可以提前阅读计算机系统基础课程的实验指南，或许能给你带来一点灵感。

指令及其相关参数

make

- -s: 在执行时不打印信息
- -f: 将跟在后面的文件作为makefile
- 其他参考: https://www.gnu.org/software/make/manual/html_node/Options-Summary.html

gcc

- -c: 只编译或汇编文件，不链接
- -ffunction-sections / -fdata-sections: 优化选项，将每个函数/数据放在它们自己的节中，链接器可以进行优化，改善空间局部性
- 其他参考: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

ld

- -e: 设置程序入口地址
- 其他参考: https://sourceware.org/binutils/docs/ld/Options.html

AM参考资料

AM概述

- 计算机系统基础课程视角 – https://nju-projectn.github.io/ics-pa-gitbook/ics2020/2.3.html
- 操作系统课程视角 – http://jyywiki.cn/OS/AbstractMachine/AM_Design

AM接口规范

- http://jyywiki.cn/OS/AbstractMachine/AM_Spec

AM选讲(习题课)

- 习题课ppt – http://jyywiki.cn/ICS/2020/slides/9.slides#/
- B站录播 – https://www.bilibili.com/video/BV1qa4y1j7xk?p=8