计算机组织与体系结构实习 Lab 2

苏睿 1600017705

一、RISC-V 工具链环境准备

下载安装 RISC-V 工具链和功能级模拟器 QEMU。

1. RISC-V 工具链

$git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

$sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev

$cd riscv-gnu-toolchain

$./configure --prefix=/opt/riscv

$make

2. 模拟器 QEMU

$ cd riscv-gnu-toolchain/qemu

$ sudo apt-get install libglib2.0-dev libpixman-1-dev

$ mkdir build && cd build

../configure --prefix=/opt/riscv --target-list=riscv32-linux-user,riscv64-linux-user,riscv32-softmmu,riscv64-softmmu

$ make

$ make install

3. 测试

运行 lab1 中使用的 ackermann 程序计算 ackermann(3, 10)

```
surui@surui:~/Documents/architecture/simulator$ /opt/riscv/bin/riscv64-unknown-elf-g++ ackermann.cpp -o ackermann
ackermann.cpp: In function 'int ackermann(int, int)':
ackermann.cpp:81:1: warning: control reaches end of non-void function [-Wreturn-type]
   81 | }
      | ^
surui@surui:~/Documents/architecture/simulator$ /opt/riscv/bin/qemu-riscv64 ackermann
8189
1.08864s
```

结果正确。

二、RISC-V 模拟器设计概述

1. 编译执行

模拟器开发环境为 Ubuntu 18.04.3 LTS，使用的编程语言为 C++，编译器为 g++7.4.0

模拟器是在课程所给的指令模拟器的基础上进行添加修改完成的，共包含 5 个文件 Read_Elf.h, Read_Elf.cpp, Reg_def.h, Simulation.cpp, Simulation.h，附有 makefile 文件，使用 make 指令即可编译获得模拟器 simulator。

模拟器执行方式

1）单指令模式 ./simulator -S [-debug] input_file，以单条指令为单位执行，若加 -debug 选项，则在每条指令执行后打印寄存器的值并等待用户输入，输入 n 单步运行打印寄存器值、输入 r 运行代码、输入 memory n pos1 pos2... posn 查看内存 n 个地址的值。

2）流水线模式 ./simulator -P input_file，以流水线模式执行，统计指令数、周期数和分支预测错误率。

2. 模拟器支持的指令集

课程所给的 RISCV-simple-greencard 全部支持，以及运行测试程序还需要用到的 addw, subw, mulw, divw, remw, slliw, srliw, sraiw。

对系统调用的支持有 PrintInt 和 PrintChar，使用汇编指令分别为 a7 寄存器赋值为 0 和 1，调用 ecall。

```
void PrintInt(int n)
{
    asm("li a7,0;" "ecall");
}

void PrintChar(char c)
{
    asm("li a7, 1;" "ecall");
}
```

3. 测试程序

测试程序位于 test_code 文件夹中，在课程所给的 test_code 基础上添加了 ackermann 函数和矩阵乘法的计算。

	add.c	#加法减法
	mul-div.c	#乘法除法
	n.c	#计算 10！
	qsort.c	#40 个数快速排序
	simple-function.c	#简单函数
	mat-mul.c	#5x5 矩阵乘法
	ackermann.c	#计算 ackermann(3, 6)　（我的模拟器好慢= =）
×	double-float	#浮点运算，本模拟器不支持

系统调用函数 PrintInt(int i)和 PrintChar(char c)定义在 syscall.c 中，编译时要一起。

编译测试程序和生成汇编使用命令：

/opt/riscv/bin/riscv64-unknown-elf-gcc -Wa,-march=rv64i -o xxx xxx.c syscall.c

/opt/riscv/bin/riscv64-unknown-elf-objdump -S xxx > xxx.S

编译完成后的可执行文件和汇编文件放在 bin_code 文件夹中。

三、模拟器实现

1. Elf 文件的解析

Read_Elf.h 和 Read_Elf.cpp 完成对 Elf 文件的解析。

1) open_file 打开输入的 Elf 文件；

2) read_Elf_header 读取 Elf 文件头，判断是否为 Elf 文件，获取程序入口地址、节数、段数、节偏移、段偏移等信息；

3) read_elf_sections 读取节头信息，获取.text 节、.data 节、.symtab 节和.strtab 节偏移；

4) read_Phdr 读取段头信息，走个形式；

5) read_symtable 读取符号表信息，获取 main 函数地址和大小，__global_pointers$全局变量指针的位置。

模拟器运行时，以上信息被记录在 elf_info.txt 文件中。

2. 模拟器模拟运行

Simulation.cpp 和 Simulation.h 模拟运行的任务。

1) load_memory 将代码和数据加载到内存中；

2) 设置 PC，全局数据段地址寄存器，栈基址寄存器；（一开始是从 elf 文件头的 entry 开始运行的，但是发现 main 函数之前好多 16 位的指令，于是就从 main 函数开始运行了，运行至 main 函数的结尾）

3) 模拟运行 run_single_inst 完成单指令执行模式，读取一条指令后，完成该指令所作的工作后，取下一条指令执行，直到结束。（本来是不必要的，主要是好写一点，还方便 debug）

流水线运行模式，采用 ICS 课程中介绍的 5 级流水线，取值、译码、执行、访存、写回，流水线寄存器定义在 Reg_def.h 中，（由于是建立在所给模板上写的，模板里定义的值我有些看不懂，所以修修改改之后就按照我的理解来了）

流水线模拟时按照写回 WB，访存 MEM，执行 EX，译码 ID，取值 IF 的顺序，以便于对数据冒险进行操作。

取值 IF，取指令放入 inst 中，记录当前指令 PC。若此时译码阶段的指令为直接跳转指令，则将该指令对应的跳转目标指令取出；若此时译码阶段的指令为条件跳转指令，则采取 always take 的策略，每次选取跳转目标作为当前指令；其余情况取下一条指令。

```
struct IFID{
    unsigned int inst;
    ULL PC;
}IF_ID,IF_ID_old;
```

译码 ID，此阶段对 IF_ID 中的指令进行译码，写入 ID_EX 中，Rd 为目的寄存器号，PC 为当前指令 PC，Imm、Reg_Rs、Reg_Rt 为执行阶段可能用到的立即数、寄存器值，is_nop 表示当前指令是否为 nop；执行阶段信息有 Ctrl_EX_ALUSrc 表示 ALU 计算操作数的来源，Ctrl_EX_ALUOp 表示 ALU 计算的操作符；访存阶段信息有 Ctrl_M_Branch 表示跳转的类型，Ctrl_M_MemWrite 表示写内存操作的类型，Ctrl_M_MemRead 表示读内存操作的类型；写回阶段信息有 Ctrl_WB_RegWrite 表示 ALU 输出写回的寄存器号，Ctrl_WB_MemtoReg 表示读取内存写回的寄存器号。

译码阶段考虑到数据冒险，若当前要取的寄存器号（如 Rs）等于访存阶段 MEM 指令的 RegWrite 目标或 MemtoReg 目标，则直接将 MEM 指令的 ALU_out 或 Mem_read 读取；若当前要取的寄存器号等于执行阶段 EX 指令的 RegWrite 目标，则将 EX 指令的 ALU_out 读取；若当前要取的寄存器号等于执行阶段 EX 指令的 MemtoReg 目标，由于该指令还未进入到访存阶段，因此需要插入一个空操作等待该指令完成访存。

```
struct IDEX{
    int Rd;
    ULL PC;
```

```
    LL Imm;
    REG Reg_Rs,Reg_Rt;
    bool is_nop;

    int Ctrl_EX_ALUSrc;
    int Ctrl_EX_ALUOp;

    int Ctrl_M_Branch;
    int Ctrl_M_MemWrite;
    int Ctrl_M_MemRead;

    int Ctrl_WB_RegWrite;
    int Ctrl_WB_MemtoReg;
}ID_EX,ID_EX_old;
```

EX 执行，此阶段根据 ID_EX 中的 ALUSrc 和 ALUOp 对数据进行对应的计算，（由于太懒了）将所有指令的周期都设置为一个周期，计算完后得到 ALU_out，其余部分保留。

```
struct EXMEM{
    ULL PC;
    int Rd;
    REG ALU_out;
    REG Reg_Rt;
    bool is_nop;

    int Ctrl_M_Branch;
    int Ctrl_M_MemWrite;
    int Ctrl_M_MemRead;
    int Ctrl_WB_RegWrite;
    int Ctrl_WB_MemtoReg;
}EX_MEM,EX_MEM_old;
```

MEM 阶段，此阶段根据 EX_MEM 中的 MemWrite 和 MemRead 进行内存的读取和写入操作，读取到的值放入 Mem_read 中。（这时我们发现 Ctrl_M_Branch 是没用的，没错！这就是照搬模板的后果）

```
struct MEMWB{
    ULL PC;
    int Rd;
    unsigned long long Mem_read;
    REG ALU_out;
    bool is_nop;
```

```
    int Ctrl_M_Branch;


    int Ctrl_WB_RegWrite;
    int Ctrl_WB_MemtoReg;


}MEM_WB,MEM_WB_old;
```

WB 阶段根据 RegWrite 和 MemtoReg 决定将 ALU_out 或 Mem_read 写入目的寄存器 Rd 中，或什么都不做。

4) 在模拟过程中若是单指令模式则记录指令数并输出；若是流水线模式则记录指令数、周期数、条件分支预测错误率并输出。

四、测试结果

1) 功能测试，对于上述 test_code 中支持的程序测试结果均正确。

如 n.c 中计算 10！结果如图：

```
surui@surui:~/Documents/architecture/simulator$ ./simulator -S bin_code/n
3628800
instuction number : 283
simulate over!
```

2) 性能测试，对于上述 test_code 中支持的程序记录结果如下：

	指令数	周期数	条件分支预测错误率
add	642	742	0.130
mul-div	667	767	0.130
n	283	317	0.100
qsort	19546	23819	0.474
simple-function	651	751	0.130
mat-mul	8492	9640	0.167
ackermann	6027180	7146445	0.667

对以上数据简要分析，add, mul-div, simple-function 都是对一个长度为 10 的数组的每个元素进行操作，程序只有运算上有差异，在数据冒险和控制冒险上表现均应相同，因此三者的 （周期数 - 指令数 = 100）且条件分支预测错误率均为 0.130，简单计算一下条件分支预测错误率，错误 3 次，共 6 + 6 + 11 = 23 次，错误率为 3 / 23 = 0.130 结果正确。

五、收获与总结

学会了 RISCV 工具链的安装使用，自己实现了一个可以用的 RISCV 模拟器，为系统调用提供了接口，复习了 ICS 中学到的流水线、数据冒险、控制冒险等内容，虽然不是很完善，但是还阔以。

（gcc 编译 shell 执行 ackermann(3,10)用 0.18s，RISCV 编译 qemu 执行 ackermann(3,10)用 1.08s，RISCV 编译我的模拟器执行 ackermann(3,6)竟然都要两分钟多，= =!）