计算机组织与体系结构实习 Lab 2

RISC-V CPU 模拟器设计与实现

一、RISC-V 工具链环境准备

下载并安装 RISC-V 工具链，git clone --recursive https://github.com/riscv/riscv-gnu-toolchain。

使用 apt-get 安装相应依赖库。

模拟器中 ELF 文件的装载使用了 boost 库，sudo apt-get install libboost-all-dev 指令完成安装。

工具链和相关依赖库下载完成后，注意到编译 toolchain 的 configure 参数除了指定安装路径--prefix = /opt/riscv，还需要加上参数--with-arch = rv64i。第一遍安装没有加上，编译出来的可执行目标文件中的 start 和 exit 代码段会出现 16 位变长指令，导致自己编写的 RISC-V 模拟器 32 位 ISA 模块无法对其进行解析。使用参数--with-arch = rv64i 后，对应的 16 位指令将由 32 位定长指令替代，模拟器方可以运行。

make 完成之后使用 export PATH = $PATH:/opt/riscv/bin 将路径添加到系统环境变量，然后就可以使用 riscv64-unknown-elf-gcc 指令编译 testbench 中的 C 语言程序为可执行文件，具体的细节见 Makefile 文件。

使用 `make package` 进行 RISC-V 指令模拟器的编译，生成可执行文件./sim

二、RISC-V 功能级模拟器

使用`./sim -f <filename> (-s)`进入 RISC-V 功能级模拟。<filename>使用编译好的./test 文件夹下的可执行文件路径，添加 -s 参数启用单步模式，否则直接输出模拟结果。

使用单步模式后，程序会将 ELF 文件代码段中的机器指令逐条解码输出，具体可用参数如下：

```
c: continue
r: print registers
x <address/hex> <size/dec>: get (size) bytes data from memory (address)
q: quit
```

输入 c 逐条打印指令；输入 r 打印当前 CPU 所有寄存器的数值（16 进制）；使用 x 并传入地址和字节大小参数，输出该物理地址中对应字节的内容，size 的大小限制在 1、2、4、8 字节之间；输入 q 退出程序。

RISC-V 模拟器的代码在./riscv-simulator 文件夹下，与功能级模拟器相关的分布在 main.c, machine.c, riscvISA.c, stage.c 中。其中 main.c 主要是定义相关的运行参数、加载 ELF 文件、开辟所需的栈空间、然后根据参数选择功能模拟器或性能模拟器。

功能级模拟器的主体定义在 machine.hpp 和 machine.cpp 中。Machine 类主要定义了寄存器模块、内存模块、指令模块、统计参量、以及相关的操作函数。其中内存模块采用简单的大数组实现，数组大小为 256MB，直接将编译好的 ELF 文件相应的 segment 加载到对应的偏移地址处，并提供对应的读写内存函数。指令模块分解成了 Fetch, Decode, Execute, MemoryAccess, WriteBack 五个阶段，在功能级实现中虽然不需要实现流水线并行，但进行分解便于实现之后的性能模拟。

riscvISA 主要是对 machine 中定义的指令模块的数据结构进行具体实现。包括了操作码、功能码、寄存器码、立即数字段等结构，以及指令类型的定义。同时通过枚举变量定义了 RISCV-simple-greencard 中需要实现的所有 RISC-V 指令类型，以及 Machine 中对应的所有寄存器名称。打印指令函数也在 riscvISA.cpp 中实现。

stage.cpp 文件主要是对 RISCV-simple-greencard 中列举的指令在 Fetch, Decode, Execute, MemoryAccess, WriteBack 五个阶段具体行为的实现。具体五个阶段的行为如下。

- Fetch：根据预测 PC 寄存器的地址从内存中取出指令
- Decode：将 32 位指令进行解析，确定是哪一条指令并取出相应参数，对于立即数做 64 位的符号拓展，将寄存器的值取出
- Execute：执行指令的计算部分

- MemoryAccess：对于需要访存的 s 与 l 指令进行访存操作
- WriteBack：需要回写寄存器的指令进行回写操作

三、RISC-V 性能模拟器

使用`./sim -f <filename> -p`进入 RISC-V 性能级模拟，性能级模拟器除了同样可以加上`-s`参数进入单步模式之外，还包括了配置文件参数和条件分支预测模拟器的参数。具体如下：

```
RISC-V Simulator Parameters::
  -f [ --filename ] arg riscv elf file
  -p [ --pipeline ]     use pipeline
  -s [ --single ]       single step mode
  -c [ --config ] arg   set config file (default 'cfg/default.cfg')
  -b [ --branch ] arg   branch prediction strategy (default 'always taken'):
                         0: always taken
                         1: always not taken
                         2: 1-bit predictor
                         3: 2-bits predictor
  -h [ --help ]         help info
```

-c 后加的配置文件的在./cfg 下，默认使用 cfg/default.cfg 文件，除此之外还有三个配置文件，分别对 32 位加法、乘除法，64 位的加法、乘除法，和访问内存的时钟周期数进行修改。-b 后的参数是选择不同的分支预测方法，0 表示总是发生跳转，1 表示总是不跳转，2 表示一位预测器，始终采取上一次的跳转策略，3 表示两位预测器，跳转与不跳转分为 WEAK 与 STRONG 两种状态，当 WEAK 状态预测错误时进入反对的 STRONG 状态。默认状态下使用总是发生跳转策略。性能级模拟器的单步模式中使用的调试指令与功能级模拟器类似，区别在于输出的指令分为 5 个流水线阶段（包含 stall 与 bubble）。具体示例如下：

```
[F] Decode 0x0a079063 (0x0000000000104a0)
[D] andi a5, a4, 15 (0x000000000001049c)
[E] --BUBBLE--
[M] --BUBBLE--
[W] bgeu t1, a2, 26 (0x0000000000010498)
```

性能模拟器相关代码分布在 main.c, machine_p.c, riscvISA.c, stage_pipe.c, config.c 中。性能模拟器的 Machine 中增加了流水线相关的寄存器，以及条件分支预测器相关类和函数，最终程序结束后打印的统计参数也与功能级模拟器不同。config.c 中包含了对模拟器可变参数的配置，目前仅包含不同指令处理器运行周期的配置，

配置文件的格式为 `str : value` 的字符串对。

stage_pipe.cpp 在 stage.cpp 的 5 个流水线阶段操作的基础上，首先在执行阶段，如果是分支指令，需要判断预测正误并更新预测 PC。然后设置了 Forwarding 的旁路，从 W/M/E 三个阶段向 D 阶段回传。在流水线结构下，主要划分了以下几个 Stall 的情况：

- Load-Use Hazard : load 指令后马上使用该寄存器，将 F/D Stall 一个周期，E 阶段插 Bubble。

- Control Hazard：分支跳转预测错误，之前的两个指令无效，D/E 设 Bubble，马上更新正确的预测 PC。

- JALR Stall : JALR 指令需要根据寄存器的值跳转，但是 F 阶段无法获得寄存器的值，因此需要等到其到 D 阶段结束获得寄存器值时才能继续流水线。

- ECALL Stall：由于模拟器的特殊性，并没有一个完整的操作系统，而 ECALL 本应陷入内核执行一系列内核指令，所以这里简单地将流水线排空，然后再执行 ECALL 指令，来近似陷入内核的现象。

需要注意的是当 Fetch 阶段 Stall 时，预测 PC 除非是分支预测错误的强制更新，否则不能更新。

四、testbench 程序运行结果

4.1 功能模拟结果

testbench 函数除了测试样例中的 add.c, double.c, mul-div.c, n!.c, qsort.c，还包括 lab1 中使用的 ackermann 函数以及矩阵乘法函数 matmul.c。在测试函数中我们添加了 RISC-V 专用的系统调用库，进行打印输出比对运行结果。

`add.c`

```
11 12 13 14 15 1 2 3 4 5

----------- Stat Info ----------
Instruction Count:    866
Running Time:         0.0002
MIPS:                 3.95
--------------------------------
```

`ackermann(3,6)`

```
To solve Ackermann(a, b), please input a:3
please input b:6
509

---------- Stat Info ----------
Instruction Count:    6113340
Running Time:         2.6994
MIPS:                 2.26
-------------------------------
```

double.c

```
---------- Stat Info ----------
Instruction Count:    17805
Running Time:         0.0022
MIPS:                 8.14
-------------------------------
```

matmul.c

```
385 935 1485 2035 2585 3135 3685 4235 4785 5335
935 2485 4035 5585 7135 8685 10235 11785 13335 14885
1485 4035 6585 9135 11685 14235 16785 19335 21885 24435
2035 5585 9135 12685 16235 19785 23335 26885 30435 33985
2585 7135 11685 16235 20785 25335 29885 34435 38985 43535
3135 8685 14235 19785 25335 30885 36435 41985 47535 53085
3685 10235 16785 23335 29885 36435 42985 49535 56085 62635
4235 11785 19335 26885 34435 41985 49535 57085 64635 72185
4785 13335 21885 30435 38985 47535 56085 64635 73185 81735
5335 14885 24435 33985 43535 53085 62635 72185 81735 91285

---------- Stat Info ----------
Instruction Count:    108777
Running Time:         0.0073
MIPS:                 14.90
-------------------------------
```

mul-div.c

```
5 10 15 20 25 1 2 3 4 5

---------- Stat Info ----------
Instruction Count:    891
Running Time:         0.0001
MIPS:                 11.00
-------------------------------
```

n!.c

```
3628800

---------- Stat Info ----------
Instruction Count:    746
Running Time:         0.0001
MIPS:                 11.30
-------------------------------
```

qsort.c

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39

---------- Stat Info ----------
Instruction Count:    21034
Running Time:         0.0015
MIPS:                 13.65
-------------------------------
```

可以得出结论，运行结果均正确，模拟器除了运行时间过短的 add 函数、浮点运算 double 函数、ackermann 递归函数，MIPS 大部分都在 10 以上。

4.2 性能模拟结果

报告中进行讨论的都在 cfg/cfg_2.cfg 文件参数设置下，其它周期参数设置下的结果可以添加参数 -c 使用其他配置文件得到。具体参数数值如下：

```
SFT_CYC: 1
ADD32_CYC: 1
ADD64_CYC: 1
MUL32_CYC: 2
MUL64_CYC: 3
DIV32_CYC: 3
DIV64_CYC: 5
MEM_CYC: 20
```

我们首先来看采用 always taken 分支预测策略的结果。

add.c

```
---------- Stat Info ----------
Instruction Count:     866
Cycle Count:           1141
Stage Count CPI:       1.32
CPU Clock CPI:         7.29

Branch Strategy:       always taken
Branch Accurancy:      62.96%   (34 / 54)
Hazard Cycle Count:    139
Stall Cycle Count:     136
-------------------------------
```

ackermann(3,6)

```
---------- Stat Info ----------
Instruction Count:     6113340
Cycle Count:           7921636
Stage Count CPI:       1.30
CPU Clock CPI:         8.25

Branch Strategy:       always taken
Branch Accurancy:      33.33%   (172246 / 516729)
Hazard Cycle Count:    1463767
Stall Cycle Count:     344529
--------------------------------
```

double.c

```
---------- Stat Info ----------
Instruction Count:     17805
Cycle Count:           21156
Stage Count CPI:       1.19
CPU Clock CPI:         2.17

Branch Strategy:       always taken
Branch Accurancy:      71.39%   (3498 / 4900)
Hazard Cycle Count:    2954
Stall Cycle Count:     397
--------------------------------
```

matmul.c

```
---------- Stat Info ----------
Instruction Count:     108777
Cycle Count:           130639
Stage Count CPI:       1.20
CPU Clock CPI:         4.76

Branch Strategy:       always taken
Branch Accurancy:      65.93%   (9369 / 14210)
Hazard Cycle Count:    18381
Stall Cycle Count:     3481
--------------------------------
```

mul-div.c

```
---------- Stat Info ----------
Instruction Count:     891
Cycle Count:           1166
Stage Count CPI:       1.31
CPU Clock CPI:         7.11

Branch Strategy:       always taken
Branch Accurancy:      62.96%   (34 / 54)
Hazard Cycle Count:    139
Stall Cycle Count:     136
--------------------------------
```

n!.c

```
---------- Stat Info ----------
Instruction Count:      746
Cycle Count:            950
Stage Count CPI:        1.27
CPU Clock CPI:          5.78

Branch Strategy:        always taken
Branch Accurancy:       56.12%   (55 / 98)
Hazard Cycle Count:     121
Stall Cycle Count:      83
-------------------------------
```

qsort.c

```
---------- Stat Info ----------
Instruction Count:      21034
Cycle Count:            27512
Stage Count CPI:        1.31
CPU Clock CPI:          8.87

Branch Strategy:        always taken
Branch Accurancy:       50.31%   (1203 / 2391)
Hazard Cycle Count:     5711
Stall Cycle Count:      767
-------------------------------
```

这里我们看到性能评估的参数主要有两个：Stage Count CPI 主要是平均每个指令需要的流水线周期，而 CPU Clock CPI 是平均每个指令执行需要的 CPU 时钟周期，该参数的数值主要与 config 文件中设置的参数大小有关，比如访存需要的时钟周期越长，CPU Clock CPI 数值越大。每个流水线周期所包含的 CPU 时钟周期也不尽相同。

可以发现，不同程序的 Stage Count CPI 大小基本都在 1.3 左右，而不同程序的 CPU Clock CPI 有较大差异。通常访存指令占比越大，CPU Clock CPI 数值越高。

接着我们统计程序在不同的分支预测策略下的预测准确率：

	Always taken	Always not taken	1-bit	2-bits
add.c	**62.96%**	37.04%	**62.96%**	**62.96%**
ackermann(3,6)	33.33%	66.67%	**99.61%**	99.42%

double.c	71.39%	28.61%	**73.55%**	58.90%
matmul.c	**65.93%**	34.07%	58.78%	58.04%
mul-div.c	**62.96%**	37.04%	**62.96%**	**62.96%**
n!.c	**56.12%**	43.88%	48.98%	46.94%
qsort.c	50.31%	49.69%	82.43%	**85.74%**

可以发现，当 Always taken 预测效果最佳时，其与 1-bit、2-bits 的预测准确率没有明显差距，而 1-bit Predictor 的预测准确率在部分程序中远远高于 Always taken 和 Always not taken，1-bit Predictor 和 2-bits Predictor 的预测准确率相近。

注：由于本学期毕业论文和其他课程任务量巨大，时间较为紧迫，以及先前对 ISA 细节尝试实现失败，lab2 的 ELF 文件加载、ISA 在 5 阶段流水线下的具体执行、一些数据统计工具部分参考了 https://github.com/EverNebula/RISC-V_Simulator 的工作，本工作主要完成了模拟器的结构设计、内存设计和逻辑框架部分，望老师和助教团队能酌情给该 lab 一些分数。后续的 lab3 将在该模拟器的基础上独立完成 cache 部分的设计工作。