体系实习lab2

蒋莹 1600017818 元培学院

一、实验环境

本机操作系统：Linux Ubuntu 4.15.0-45-generic （16.04 LTE）

编译器：gcc/g++ version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10))

1. RISC-V工具链环境准备

Github上下载速度较慢，采用了教学网提供的freedom/rocket-chip/riscv-tools 安装。实验基于RISC-V rv64I（64位基本指令系统），所以添加配置--with-arch=rv64i，防止生成没有进行模拟的其它riscv指令。

```
mkdir build
cd build
../configure --with-arch=rv64i --prefix=/path/to/toolchain
make -j$(nproc)
```

最后将[/path/to/toolchain]/bin 加入环境变量，写入~/.bashrc文件即可使用。

```
export PATH=/path/to/toochain/bin/:$PATH
```

将c文件编译为riscv ELF文件：使用-Wa,-march=rv64i选项让编译器针对RV64I标准指令集生成ELF程序。

```
riscv64-unknown-elf-gcc -Wa,-march=rv64i mylib.c matrix.c -o matrix.riscv
```

使用objdump将ELF文件反汇编，生成文本文件：

```
riscv64-unknown-elf-objdump -d matrix.riscv > matrix.asm
```

二、设计概述

模拟器实现了所有rv64i的指令。主要模块包装成类，包括：内存模拟类memory、cpu模拟类sim、流水线寄存器模拟类X_REG等。

sim类中包含了加载文件和初始化模拟器的过程，并通过一个大的主循环模拟每个始终周期流水线处理指令的的过程：取指、译码、执行、访存、写回。这五个过程的定义基本和ICS课程定义一致，任务都是由当前寄存器的值生成下一周期寄存器的值及全部控制信号。其中取指阶段进行本阶段PC的选择和下一PC的预测。译码阶段解析指令含义并负责来自其它阶段的转发的选择。执行阶段进行指令对应的算术操作。需要访问内存的指令再访存阶段访问内存。写回阶段将新的值写入寄存器。当模拟过程正常执行完毕后退出并打印数据。

在单个周期内五个阶段的执行顺序为写回、访存、执行、译码、取指。硬件执行在下一个时钟周期到来之前，当前周期的组合逻辑的结果已经较早计算得到，因此当前周期所需的控制信号得到满足。而软件模拟为了生成所有所需的控制和转发信号，需要倒序执行。

五个流水线寄存器实现为控制信号（bubbe、stall），两套寄存器值域。一套作为本阶段输出，另一套接受来自上一阶段的输入作为下一阶段的新输出。某些值域对书中定义进行了合并和简化便于书写（比如M阶段的valE和valM合并，因为二者最多使用一个）。

memory类包含内存访问的接口，接受地址、请求长度、写入的值等参数。在读写之前先检查地址合法性，再进行访问。支持打印内存空间的值。

此外将elf文件装载内存的过程封装为load函数，其中使用了elfio工具：http://elfio.sourceforge.net/elfio.pdf；并将riscv的print的系统调用接管为打印字符串、32位整数、字符三个函数，分别为print_s, print_d, print_c。

1. 代码结构

在根文件夹下有如下文件：

```
$ tree
.
├── elfio
│   ├── elfio_dump.hpp
│   ├── elfio_dynamic.hpp
│   ├── elfio_header.hpp
│   ├── elfio.hpp
│   ├── elfio_note.hpp
│   ├── elfio_relocation.hpp
│   ├── elfio_section.hpp
│   ├── elfio_segment.hpp
│   ├── elfio_strings.hpp
│   ├── elfio_symbols.hpp
│   ├── elfio_utils.hpp
│   └── elf_types.hpp
├── memory.txt
├── Read_Elf.cpp
├── sim.out
├── simulation.cpp
├── Simulation.h
└── test-sample
    ├── ack.c
    ├── ack.riscv
```

```
    ├── matrix.c
    ├── matrix.riscv
    ├── mylib.c
    ├── mylib.h
    ├── mytest.c
    ├── mytest.riscv
    ├── qs.c
    ├── qs.riscv
    └── testing-samples-given
        ├── add.c
        ├── add.riscv
        ├── double-float.c
        ├── mul-div.c
        ├── mul-div.riscv
        ├── n!.c
        ├── n!.riscv
        ├── qsort.c
        ├── qsort.riscv
        ├── simple-fuction.c
        └── simple-function.riscv
```

主要代码由于命名空间的不同分成了Read_elf.cpp文件和simulation.cpp文件。Simulation.h包含宏定义和类定义。elfio包在elfio文件夹中。模拟器的入口为main函数，在simulation.cpp中。test-sample中有要求的三个测试程序，其中的函数调用均是库mylib.c接管的系统调用。

2. 编译运行

编译时需要使用elf文件夹中的头文件。

```
g++ -I./ *.cpp -o ./sim.out -std=c++11
```

得到sim.out，是模拟器的可执行文件。运行方法：

```
./sim.out [riscv-elffile] [-m] [-v] [-s] [-b strategy]
```

riscv-elffile为待执行的riscv ELF文件。-m为运行结束时打出所有内存空间到memory.txt。-v为verbose，会详细输出模拟器流水线每一阶段步骤。-s为单步调试。-b添加分支预测策略，模拟器支持AT(Always Taken), NT(Not Taken), BTFNT(Backward Taken, Forward Not Taken)。

不加任何参数，会默认用AT执行程序，打印程序输出。正常退出前打印CPU统计数据。如下：

```
$ ./sim.out test-sample/ack.riscv

...(output of program)
Exit normally
Cycles: 120679
Instructions: 99051
CPI: 1.218352
Load-use hazards: 9707
Jumps: 8412
Mispredicts: 1754
Control hazards (JXX + BXX): 10166
AT prediction accuracy: 0.652467
```

单步模式-s，包含详细的步骤输出，每一步可选打印寄存器的值和内存的值。典型输出如下：

```
$ ./sim.out test-sample/ack.riscv -s
WB: Bubble
MEM: Bubble
EX: Bubble
ID: 00003197 interpreted as 'AUIPC GP 3'
IF: Fetch instruction 1a018193 at address 100b4
IF: predicted PC 000100b8 for next clock
'g' to proceed, 'm' to dump memory and 'r' for reg info
g
WB: Bubble
MEM: Bubble
EX: AUIPC
ID: 1a018193 interpreted as 'ADDI GP GP 416'
EX Forward GP to ID val1
IF: Fetch instruction 81818513 at address 100b8
IF: predicted PC 000100bc for next clock
'g' to proceed, 'm' to dump memory and 'r' for reg info
m
Memory dump in memory.txt
r
Register info
PC 100bc
ZERO: 0 (0)
RA: 0 (0)
SP: 7fffc000 (2147467264)
GP: 0 (0)
TP: 0 (0)
T0: 0 (0)
T1: 0 (0)
...
```

三、具体设计和实现

1. 可执行文件的装载

ELF文件解析采用了开源库ELFIO(https://github.com/serge1/ELFIO)。这个库包含c++头文件，在编译时制定库的搜索路径-I即可。它提供了读取elf文件各个部分的API，比如节和段的内容读取等。详见Read_Elf.cpp。利用API可以直接按照elf文件制定的大小分配内存和写入数据。使用API直接读取entry入口后设置pc寄存器即可开始执行。

```
void loader(elfio &reader, Memory &memory)
{
  Elf_Half n = reader.segments.size();
  for (int i = 0; i < n; i++)
  {
    segment *pseg = reader.segments[i];
    uint32_t data_size = pseg->get_file_size(), seg_size = pseg-
>get_memory_size(), seg_start = pseg->get_virtual_address();
    for (uint32_t p = seg_start; p < seg_start + seg_size; p += 4096)
    {
      if (!memory.isValidAddr(p))
        memory.alloc(p);
    }
    for (uint32_t p = seg_start; p < seg_start + data_size; p++)
      memory.store_byte(p, pseg->get_data()[p - seg_start]);
  }
}
```

2. 内存组织方式

内存管理使用了一个类Memory，提供内存访问接口，详见Simulator.h头文件中的Memory类。受到展示的同学的启发采用二级页表的目录组织方式来直接映射地址（虚存作为“实存”），避免了地址转换引起bug。32位地址空间可以用uint_32t表示。把地址划分为大小为4096字节的页，这样可以正好划分为二级页表，每一级页表索引1024个子页。虚拟地址的最高10位作为页目录的索引，次高10位作为二级页表的索引，最后12位作为页内偏移。为了便捷把页目录和页表项的计算都用宏定义完成。示例如下：

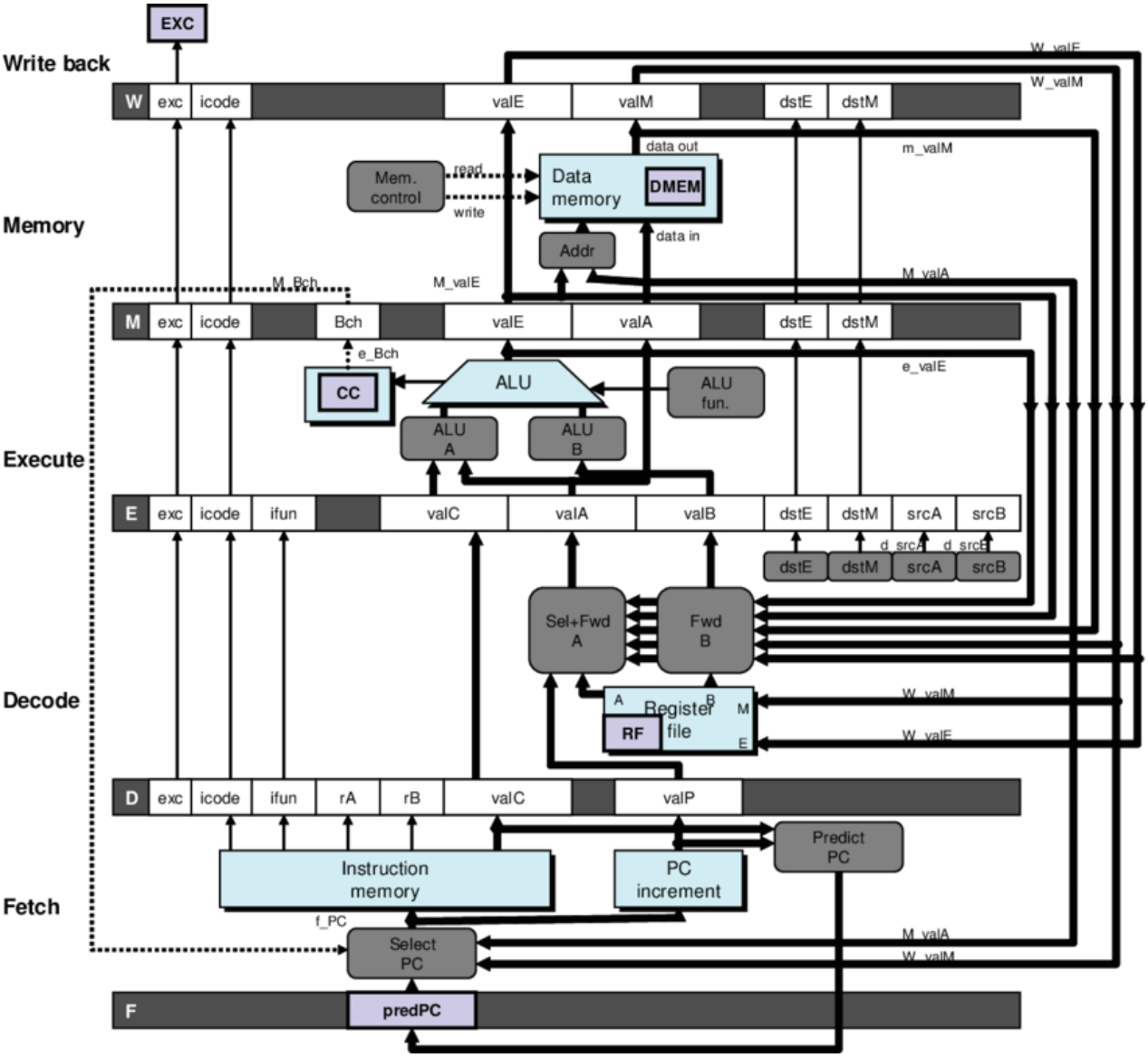
```
#define PDE(addr) ((addr >> 22) & 0x3FF)
#define PTE(addr) ((addr >> 12) & 0x3FF)
#define OFF(addr) (addr & 0xFFF)

void store_byte(uint32_t addr, unsigned char val)
{
  uint32_t i = PDE(addr);
  uint32_t j = PTE(addr);
  uint32_t k = OFF(addr);
  if (!isValidAddr(addr))
```

```
  {
    printf("invalid write at %x\n", addr);
    exit(-1);
  }
  memory[i][j][k] = val;
}
bool store(uint32_t addr, uint64_t val, int bytes)
{
  for (int i = 0; i < bytes; i++)
    store_byte(addr + i, (val >> 8 * i) & 0xFF);
  return true;
}
```

3.3 指令语义的解析和控制信号的处理

下面是ICS上Y86-32五个处理器阶段的设计，我的实现基本是下图的简化版本。寄存器的设计见Simulation.h，解析和处理见simulation.cpp。图中icode和ifun作用对应riscv中的opcode和func7/func3。五个阶段代码在simulation.cpp对应的五个函数中。下面会说明具体的修改：



1. IF

IF阶段分为三部分：

1. Select PC

选取F阶段上一周期的结果predictPC、跳转指令JXX进入E阶段转发回来的计算好的PC e_calcPC、分支指令BXX预测错误后转发回来的not taken branch（E阶段在上一周期发现错误后，本周期从M转发回来作为新的PC）。

2.访问内存取指

3.Predict PC

预测默认PC+4，因为RV64I全部指令为32位长。如果刚取出来的是跳转指令则根据分支预测策略进行跳转。为了简便我把每个阶段没有选取的分支地址也记录在指令信息中随流水线向前进行。

2. ID

1.指令解析

根据RISCV-ISA-list.pdf对指令进行解析。为了debug方便没有使用硬件中的多选器。OPcode使用宏定义为绝对无符号数值，从而可以作为switch条件。取出每条指令的OPcode经过switch判断，然后对于有的func3的指令作为下一级别switch判断条件，如果有func7同理。最终获得所需的valA，valB，偏移量等准备进行运算。注意所有PC、内存地址和指令编码均为uint_32t；即除了操作数、偏移量、符号扩展读取的操作所有涉及流水线内容都是用无符号数。

2.接受寄存器值的转发

由于五个阶段是倒序执行的，所以天然避免了W阶段的所有转发。为了优先级，先检测M阶段，后检测E阶段转发。如果都为真，值会来自E阶段较新执行的指令。检测条件都是如果被转发阶段要写寄存器，写的不是0寄存器（有指令比如ret会被翻译为jalr ZERO 0 XX，语义是不希望写入任何寄存器），且要写的寄存器是当前D阶段中指令需要读取的寄存器之一。M和E阶段都最多写入一个寄存器，所以把Y86中M阶段的的valE和valM合并为valM，只需要各检查这一个待写入的值即可。总共需要检测的信号为：m_valM和e_valE、m_reg_write和e_reg_write。

```
int destReg = Mreg.cur.destReg;
if (m_reg_write && destReg != REG_ZERO)//ret is JALR ZERO 0 A0
{
  if (reg1 == destReg)
  {
    valA = m_valM;
    if (verbose)
      printf("MEM Forward %s to ID valA\n", reg_name[destReg]);
    forw = true;
  }
  if (reg2 == destReg)
  {
    valB = m_valM;
    if (verbose)
```

```
      printf("MEM Forward %s to ID valB\n", reg_name[destReg]);
    forw = true;
  }
}
```

3. EX

根据指令语义，将上一阶段确定的valA/B进行对应的算术或逻辑运算，同时也计算下面的转发信号。系统调用和库函数接口的处理见下一部分。

```
bool e_isJ = 0, e_Bwrong = 0, e_reg_write = 0, m_reg_write = 0;
int64_t e_valE = 0, m_valM = 0;
uint64_t e_calcPC = 0;
```

4. MEM

执行内存的有符号/无符号读写操作，并计算上述转发值。

5. WB

将执行结果写回寄存器。由于最先执行，需要写回就直接写回寄存器了，就不需要写转发了。

6. 主函数对冒险的处理

冒险处理同ICS的处理方式，分为下面三类。每次在5阶段运行完毕后统一判断。在冒险冲突时处理方式也和ics一致：设置bubble优先级高于stall即可。

在F阶段等待正确的转发时bubble和stall效果相同。后两种控制的冒险经测试也可以在F使用bubble。

冒险	F	D	E	M	W
Load-Use Hazard	stall	stall	bubble	-	-
JAL, JALR	stall	bubble	-	-	-
Mispredicted Branch	stall	bubble	bubble	-	-

3.4 系统调用和库函数接口的处理

使用自定义的函数接口内嵌汇编进行ecall系统调用。编译器默认行为在系统调用时用A0保存函数参数，A7保存系统调用号。如果E阶段syscall添加93号exit调用，return 0指令可以被成功执行。所以我们在库函数里只需要接管打印字符串、32位整数、字符的三个操作，分别为print_s, print_d, print_c。

```
void print_s(char *str) // 参数在a0
{
    asm("li a7, 0;"
        "scall");
};
```

```
void print_c(char c)
{
    asm("li a7, 1;"
        "scall");
};

void print_d(int d)
{
    asm("li a7, 2;"
        "scall");
};
```

3.5 性能计数

代码量总量不大，我设置了如下全局变量来计数。

```
int inst_cnt = 0;
int cycle_cnt = 0;
int correct_predict = 0;
int mispredict = 0;
int cnt_j = 0;
int control_cnt = 0;
int loaduse_cnt = 0;
```

其中指令总数计数在M阶段排除bubble后进行，因为M阶段不会被Stall且可以进入的不会是错误指令，也保证了不同策略时总指令数相同。cycle综述即流水线循环次数。三种冒险在对应判断条件触发时记录。

3.6 调试接口

译码和执行阶段相对复杂，主要采取输出调试单独debug。OPcode使用宏定义，并重新反向映射(int 1~54)到对应的字符串便于输出调试。0号指令的宏留空，防止和bubble的结果混淆。0号寄存器特判，因为会有ret这样的指令写0号寄存器。如果使用-v执行就可以看到所有流水线每一步决策输出，包括decode结果。

而对于逻辑调试，还是输出调试大法好，干瞪眼是没用用的。哪怕输出乍一看无头绪，只要输出每一个阶段决策的结果，会大大提升debug效率。单步调试需要交互比较麻烦，一般使用-v输出所有流水线结果并重定向到文件，同时结合objdump结果来调试。

举例：我发现我的模拟器计算矩阵乘法总是出错，结果总固定的是标准答案的3倍。而指令集并不包含mul指令，反汇编发现每一个乘法使用移位和加法循环实现。阅读可知这个操作有一个最多32次的循环，参数在a0和a1，根据被乘数a1每一位是0是1来进行乘法，也就是用循环模拟乘法器实现原理。

```
00000000000102fc <__muldi3>: #op1在a0，op2在a1，a0存放返回值
   102fc: 00050613            mv  a2,a0
   10300: 00000513            li  a0,0 # addi A0 ZERO 0
   10304: 0015f693            andi  a3,a1,1
   10308: 00068463            beqz  a3,10310 <__muldi3+0x14> # beq A3 ZERO 8 -
--srli
   1030c: 00c50533            add a0,a0,a2
   10310: 0015d593            srli  a1,a1,0x1 # srli A1 A1 1
   10314: 00161613            slli  a2,a2,0x1 # slli A2 A2 1
   10318: fe0596e3            bnez  a1,10304 <__muldi3+0x8> # bne A1 ZERO -20
---andi
   1031c: 00008067            ret
   10320: 0000                unimp
  ...
```

最后选取带有发生计算错误的指令地址(1030c)定位到模拟器输出，根据输出发现add A0 A0 A2总是被连续取指3次，而此前刚好是分支预测错误。所以具体打印分支预测阶段判断条件，发现流水线遇到bubble后直接选择传递bubble到下一阶段，而"不执行有意义的操作"直接返回，导致某一信号没被更新。所以设置bubble后，必须在进入bubble的下一个周期内单独把当前阶段的流水线寄存器输出和转发信号归0。此外还要注意不能使用流水线下一阶段的输入作为判断条件，因为这一输入下一阶段不一定成立，可能被bubble。

四、功能测试和性能评测

1. 运行5个定点程序，给出动态执行的指令数，执行周期数，平均CPI。

编译方法见第一部分。均采用AT策略。

给定的五个程序结果如下：

add.c

```
$ ./sim.out test-sample/testing-samples/add.riscv
Exit normally
Cycles: 1167
Instructions: 909
CPI: 1.283828
Load-use hazards: 58
Jumps: 39
Mispredicts: 80
Control hazards (JXX + BXX): 119
AT prediction accuracy: 0.534884
```

n!.c

```
$ ./sim.out test-sample/testing-samples/n\!.riscv
Exit normally
Cycles: 1444
Instructions: 1128
CPI: 1.280142
Load-use hazards: 34
Jumps: 73
Mispredicts: 104
Control hazards (JXX + BXX): 177
AT prediction accuracy: 0.539823
```

qsort.c

```
$ ./sim.out test-sample/testing-samples/qsort.riscv
Exit normally
Cycles: 24842
Instructions: 19483
CPI: 1.275060
Load-use hazards: 3174
Jumps: 310
Mispredicts: 937
Control hazards (JXX + BXX): 1247
AT prediction accuracy: 0.532901
```

simple-function.c

```
$ ./sim.out test-sample/testing-samples/simple-function.riscv
Exit normally
Cycles: 1179
Instructions: 919
CPI: 1.282916
Load-use hazards: 58
Jumps: 41
Mispredicts: 80
Control hazards (JXX + BXX): 121
AT prediction accuracy: 0.534884
```

mul-div.c

```
$ ./sim.out test-sample/testing-samples/mul-div.riscv
Exit normally
Cycles: 1192
Instructions: 934
CPI: 1.276231
Load-use hazards: 58
Jumps: 39
Mispredicts: 80
Control hazards (JXX + BXX): 119
AT prediction accuracy: 0.534884
```

2. 运行三个接管系统调用的程序，给出动态执行的指令数，执行周期数，平均CPI。

自己编写的三个测试程序见下。已经打出了原始数据。

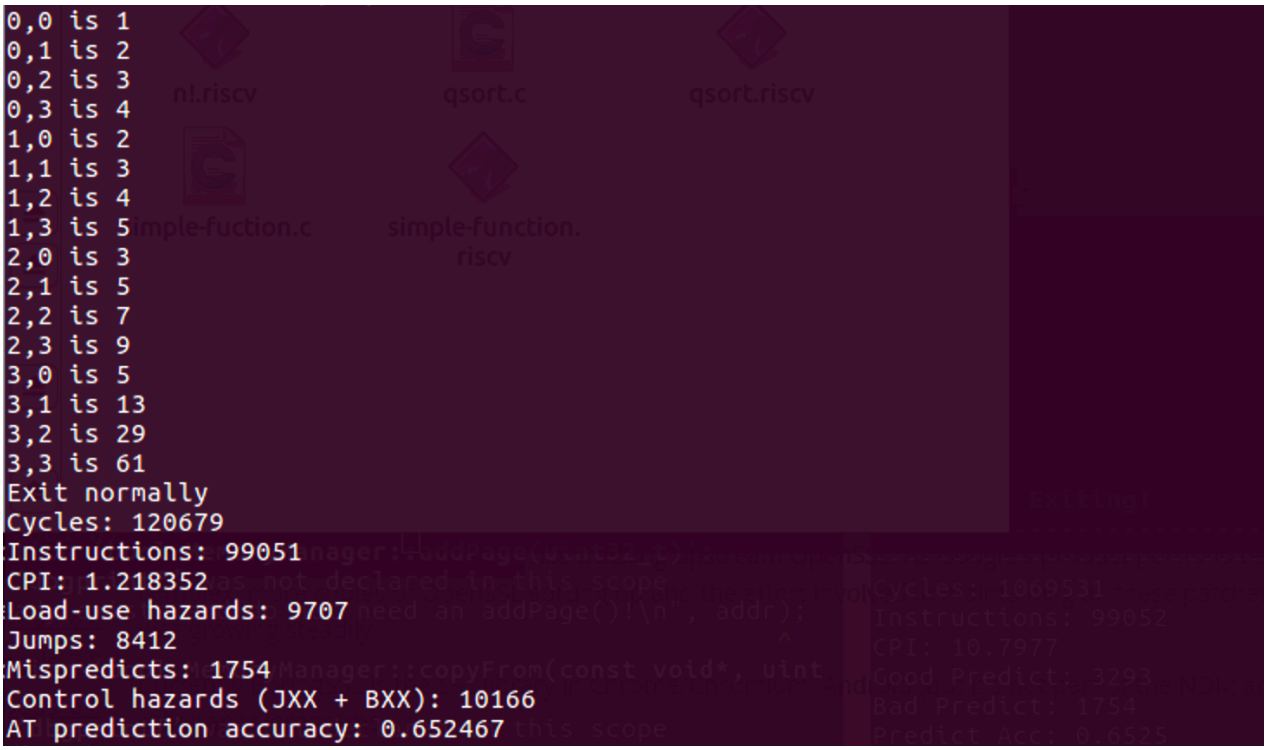
qs.c

```
$ ./sim.out test-sample/qs.riscv
before:
12 85 25 16 34 23 49 95 17 61 12 85 25 16 34 23 49 95 17 61 12 85 25 16 34 23 49
 95 17 61 12 85 25 16 34 23 49 95 17 61 12 85 25 16 34 23 49 95 17 61 12 85 25 1
6 34 23 49 95 17 61 12 85 25 16 34 23 49 95 17 61 12 85 25 16 34 23 49 95 17 61
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
after:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 16 16 16 16 16 1
6 16 16 17 17 17 17 17 17 17 17 23 23 23 23 23 23 23 23 25 25 25 25 25 25 25 25
34 34 34 34 34 34 34 34 49 49 49 49 49 49 49 49 61 61 61 61 61 61 61 61 85 85 85
 85 85 85 85 85 95 95 95 95 95 95 95 95
Exit normally
Cycles: 43778
Instructions: 33916
CPI: 1.290777
Load-use hazards: 5682
Jumps: 1387
Mispredicts: 1396
Control hazards (JXX + BXX): 2783
AT prediction accuracy: 0.526298
```

matrix.c

```
$ ./sim.out test-sample/matrix.riscv
arr1
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
arr2
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
arr3
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
0 45 90 135 180 225 270 315 360 405
Exit normally
Cycles: 94497
Instructions: 76845
CPI: 1.229709
Load-use hazards: 8692
Jumps: 3471
Mispredicts: 2744
Control hazards (JXX + BXX): 6215
AT prediction accuracy: 0.622091
```

ack.c



3.统计数据

Program	Cycles	Instructions	CPI	Load-Use	Jumps	Mispredicted B.
add.c	1167	909	1.283	58	39	80
n!.c	1444	1128	1.280	34	73	104
qsort.c	24842	19483	1.275	3174	310	937
simple-function.c	1179	919	1.282	58	41	80
mul-div.c	1192	934	1.276	58	39	80
qs.c	43778	33916	1.290	5682	1387	1396
matrix.c	94497	76845	1.229	8692	3471	2744
ack.c	120679	99051	1.218	9707	8412	1754

对于所有有效指令，平均CPI在1.27左右，没有偏离1太远，说明流水线功能较为高效。具体值依赖于程序本身冒险的多少和分支预测策略的性能。

4. 分支预测策略比较

对于较大的三个程序使用分支预测，得到统计：

Program	Always Taken	NT	BTFNT
qs.c	0.526298	0.473702	**0.762131**
matrix.c	0.622091	0.377909	**0.685167**
ack.c	**0.652467**	0.347533	0.363186

BTFNT本身针对循环来设计，默认循环执行多次时会获得较高性能。所以在循环较多的快速排序（需要检查每个元素）和矩阵乘法（需要迭代每个结果）中表现较好，当然AT在这种情况也会比较占优。对于ACK函数，其中只有一个大循环语句和大量ifelse。由于函数运算本身复杂，所以循环次数少，BTFNT结果不明显。而由于值普遍比较小，递归后=0的为真判断就会比较多，所以AT的优势就出来了。

五、其它需要说明的内容

1. 意见和建议

1. 学习到了debug的时候输出大法好。
2. 作业思考量和debug量都不小，很锻炼人。课上增加同学的分享对我的帮助很大，然而有bug的时候非常绝望，希望可以再放宽些时间来做……
3. 希望材料可以更新一些，有些文档比如greencard上错误较多，必须依赖更全的其它文件才可以。