实验-优化Y86-64流水线处理器性能

班级： 07812201 学号： 1820221050 姓名： 丘绎楦

一、实验目的

在本实验中，将了解流水线Y86-64 处理器的设计和实现，优化它和基准程序以最大限度地提高性能。

二、实验内容

实验分为三个部分，每个部分都有自己的上交成果。

第一部分（Part A）：您将编写一些简单的Y86 64 程序，并熟悉Y86-64工具。第二部分（Part B）：您将使用新的指令扩展SEQ模拟器。

第三部分（Part C）：在前两部分的基础上，优化Y86-64基准测试程序和流水线处理器设计

三、实验步骤

环境准备：本次实验使用VMWARE WORKSTATION PRO 17 的Ubuntu 18.04.6 系统上进行的。

在乐学平台上下载sim.tar，并把文件拷贝到虚拟机中。并使用tar -xvf sim.tar进行解压。然后再 sim 目录执行make clean;make 指令构建工具。进行make时提示缺少依赖，需安装以下依赖：

sudo apt install tcl tcl-dev tk tk-dev

sudo apt install flex

sudo apt install bison

第一部分（Part A）：

在这一部分中的任务将在sim/misc目录下工作，在此需要完成3个汇编程序（sum.ys, rsum.ys, copy.ys），分别完成examples.c的3个函数。

编写完三个汇编程序后，执行./yas sum.ys生成sum.yo文件，同理对其余两个一样。之后执行./yis sum.yo 查看运行结果。

第二部分（Part B）：

在这一部分中的任务将在/sim/seq目录下工作，任务是扩展SEQ处理器以支持iaddq指令，在此我们只需要修改 seq-full.hcl 文件。在该文件中可以看到# Instruction code for iaddq instruction

wordsig IIADDQ 'I\_IADDQ'

符号表已经添加了iaddq，这时我们需要修改 Fetch 阶段的以下部分，增加IADDQ指令：

bool instr\_valid = icode in

{ INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,

IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };

bool need\_regids =

icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,

IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

bool need\_valC =

icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };

修改 Decode阶段的部分：

word srcB = [

icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : rB;

icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;

1 : RNONE; # Don't need register

];

word dstE = [

icode in { IRRMOVQ } && Cnd : rB;

icode in { IIRMOVQ, IOPQ, IIADDQ} : rB;

icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;

1 : RNONE; # Don't write any register

];

修改 Execute 阶段的部分：

word aluA = [

icode in { IRRMOVQ, IOPQ } : valA;

icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;

icode in { ICALL, IPUSHQ } : -8;

icode in { IRET, IPOPQ } : 8;

# Other instructions don't need ALU

];

word aluB = [

icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,

IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;

icode in { IRRMOVQ, IIRMOVQ } : 0;

# Other instructions don't need ALU

];

bool set\_cc = icode in { IOPQ, IIADDQ };

修改完成后，需要基于此HCL文件构建SEQ模拟器（ssim）的新实例，然后对其进行测试：

执行 make VERSION=full；

出现问题报告找不到 tk.h 头文件，解决方法是把Makefile中第20行的tcl版本改为8.6、第26行改为CFLAGS=-Wall -O2 -DUSE\_INTERP\_RESULT。再次执行还是报错undefined reference to `matherr'，此时需要修改/sim/pipe/psim.c的806、807行和/sim/seq/ssim.c的844、845行，即含有matherr的一行和下一行。然后便能编译成功，忽略Warning信息。

执行

./ssim -t ../y86-code/asumi.yo

(cd ../y86-code; make testssim)

(cd ../ptest; make SIM=../seq/ssim)

(cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)

来进行测试，如果所有测试都能顺利通过，那么此部分就完成了。

第三部分（Part C）：

这部分的任务在/sim/pipe目录下工作，任务是修改pipe-full.hcl和ncopy.ys文件，使ncopy.ys尽可能快速地运行。

首先对pipe-full.hcl进行修改，如Part B部分，扩展iaddq指令功能。

然后优化ncopy.ys的代码，主要思想是采用循环展开进行优化，把循环展开成8x8，然后最后不足8个时利用二分查找，判断剩余多少个，然后跳转到剩余不同数的展开。

ncopy.ys:

iaddq $-8, %rdx

jl Test

Loop8x8:

#取地址

mrmovq 0(%rdi), %r8

mrmovq 8(%rdi), %r9

mrmovq 16(%rdi), %r10

mrmovq 24(%rdi), %r11

mrmovq 32(%rdi), %r12

mrmovq 40(%rdi), %r13

mrmovq 48(%rdi), %r14

mrmovq 56(%rdi), %rbx

#赋值

rmmovq %r8, 0(%rsi)

rmmovq %r9, 8(%rsi)

rmmovq %r10, 16(%rsi)

rmmovq %r11, 24(%rsi)

rmmovq %r12, 32(%rsi)

rmmovq %r13, 40(%rsi)

rmmovq %r14, 48(%rsi)

rmmovq %rbx, 56(%rsi)

#判断是否可以count+1

judge0:

andq %r8, %r8

jle judge1

iaddq $1, %rax

judge1:

andq %r9, %r9

jle judge2

iaddq $1, %rax

judge2:

andq %r10, %r10

jle judge3

iaddq $1, %rax

judge3:

andq %r11, %r11

jle judge4

iaddq $1, %rax

judge4:

andq %r12, %r12

jle judge5

iaddq $1, %rax

judge5:

andq %r13, %r13

jle judge6

iaddq $1, %rax

judge6:

andq %r14, %r14

jle judge7

iaddq $1, %rax

judge7:

andq %rbx, %rbx

jle Loop

iaddq $1, %rax

Loop:

iaddq $64,%rdi

iaddq $64,%rsi

iaddq $-8,%rdx

jge Loop8x8

Test:

iaddq $5,%rdx

jl LeftChild

je Remain3

RightChild:

iaddq $-2,%rdx

jg RightChild2

jl Remain4

je Remain5

LeftChild:

iaddq $2,%rdx

je Remain1

jl Done

iaddq $-1,%rdx

jmp Remain2

RightChild2:

iaddq $-1,%rdx

je Remain6

Remain7:

mrmovq 48(%rdi), %r8

rmmovq %r8,48(%rsi)

andq %r8,%r8

Remain6:

mrmovq 40(%rdi), %r8

jle Extra6

iaddq $1, %rax

Extra6:

rmmovq %r8, 40(%rsi)

andq %r8, %r8

Remain5:

mrmovq 32(%rdi), %r8

jle Extra5

iaddq $1, %rax

Extra5:

rmmovq %r8, 32(%rsi)

andq %r8, %r8

Remain4:

mrmovq 24(%rdi), %r8

jle Extra4

iaddq $1, %rax

Extra4:

rmmovq %r8, 24(%rsi)

andq %r8, %r8

Remain3:

mrmovq 16(%rdi), %r8

jle Extra3

iaddq $1, %rax

Extra3:

rmmovq %r8, 16(%rsi)

andq %r8, %r8

Remain2:

mrmovq 8(%rdi), %r8

jle Extra2

iaddq $1, %rax

Extra2:

rmmovq %r8, 8(%rsi)

andq %r8, %r8

Remain1:

mrmovq (%rdi), %r8

jle Extra1

iaddq $1, %rax

Extra1:

rmmovq %r8, (%rsi)

andq %r8, %r8

jle Done

iaddq $1, %rax

修改完成后

执行

make drivers （每次修改ncopy.ys时执行）

make psim VERSION=full （每次修改pipe-full.hcl时执行）

../misc/yis sdriver.yo （检测ncopy.ys与YIS一起正常工作）

./correctness.pl （检测正确率）

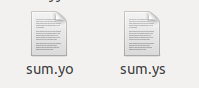
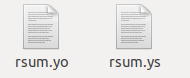
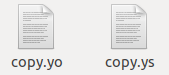
./benchmark.pl （检测代码性能得分）

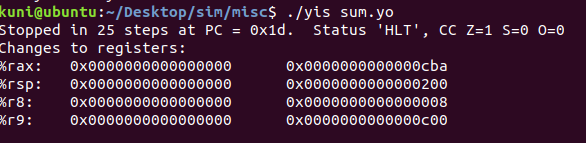
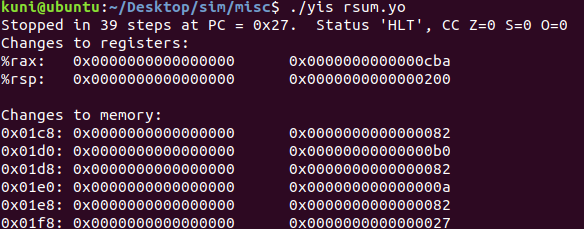
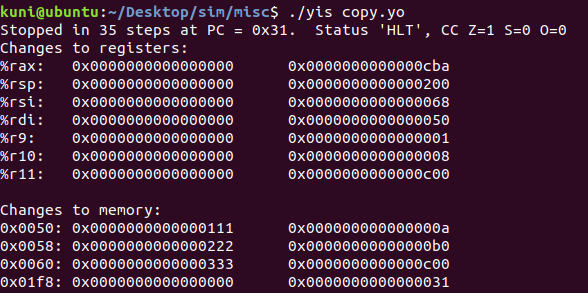
四、实验结果及分析

第一部分（Part A）：

编写完三个汇编程序后，进行编译：

生成对应的.yo文件：

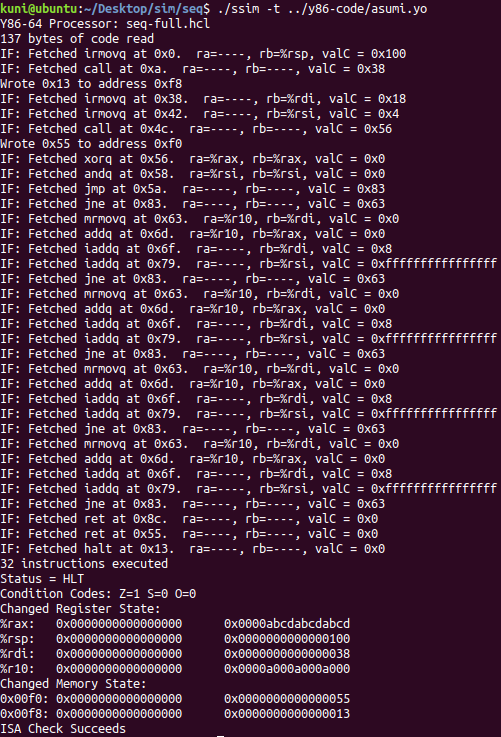


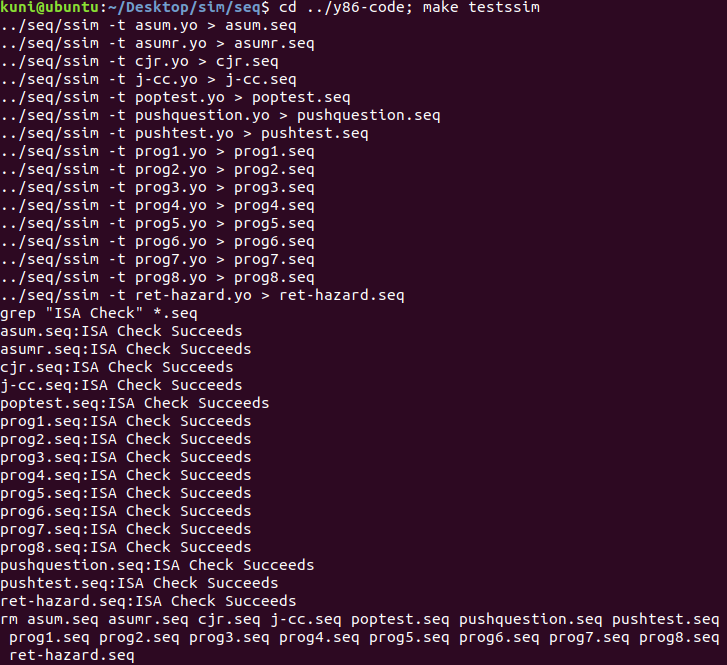
执行./yis 对应文件查看运行结果：

可以看到%rax的值是0xcba，程序运行结果正确，并且各个栈变化也可以看出结果的正确性。

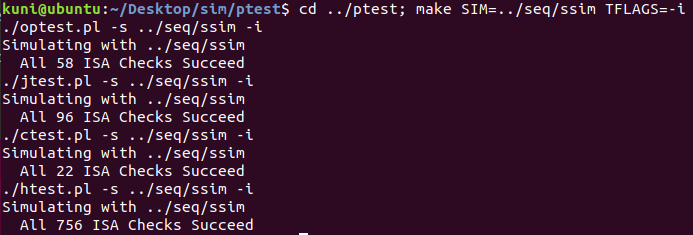
第二部分（Part B）：

在修改完seq-full.hcl文件后，进行检测的结果：

执行./ssim -t ../y86-code/asumi.yo

执行(cd ../y86-code; make testssim)

执行(cd ../ptest; make SIM=../seq/ssim)

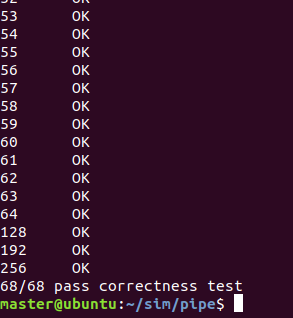
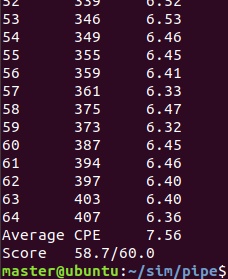
执行(cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)

第三部分（Part C）：

修改完pipe-full.hcl和ncopy.ys文件后，进行代码的正确性和性能进行检测。

可以看到

正确性为百分百正确

性能Average CPE为7.56，分数为58.7/60

五、实验收获与体会

通过本次实验，我对Y86-64处理器的架构和实现有了更深刻的理解，并且对程序代码优化技巧有了进一步的认识。通过亲手编写多段汇编代码，我不仅加深了对汇编语言的掌握，还学会了如何高效地运用它。特别是在性能优化方面，我通过实践循环展开技术，深刻体会到了它在提升代码效率方面的显著作用。

附录：程序清单及说明

sum.ys

rsum.ys

copy.ys

seq-full.hcl

ncopy.ys

pipe-full.hcl