CSAPP实验4:Arch

实验准备

实验环境：Ubuntu 20.04 LTS 64位 in Vmware Workstation 16 Player

按照指导手册安装相关依赖，并在Makefile中添加 `GUIMODDE=-DHAS_GUI` 以启用GUI模式

因为老师给出的部分指导不准确，在这里我给出一些补充和修改：

我在做实验时，在sim目录下的Makefile添加或修改参数"CFLAGS"是起不到作用的，需要去手动更改pipe，seq和y86-code目录下的Makefile才可以生效

Part A

在这个part中我们的工作目录是misc，要写3个汇编程序，分别完成examples.c中对应的3个函数

sum.ys

这道题就是让人熟悉怎么写Y86-64汇编程序的，没有难度，只需要注意一个节点的大小是16字节，从字段"val"到字段"next"需要加8个字节。直接贴代码（寄存器%rdi保存参数，%rax保存返回值）：

```
# name : Bowen Liu
# ID : 1120201883
# date : 2022.11.12

.pos 0
    irmovq stack,%rsp
    irmovq ele1,%rdi
    call sum
    halt

.align 8
ele1:
    .quad 0x00a
    .quad ele2
ele2:
    .quad 0x0b0
    .quad ele3
ele3:
    .quad 0xc00
    .quad 0

sum:
    irmovq $0,%rax
    irmovq $8,%r8
    andq %rdi,%rdi
    jmp test
```

```
loop:
    mrmovq (%rdi),%r9
    addq    %r9,%rax
    mrmovq  8(%rdi),%rdi
    andq %rdi,%rdi


test:
    jne loop
    ret

.pos 0x200
stack:
```

运行结果：

```
root@ubuntu:/home/lbw/sim/misc# ./yis sum.yo
Stopped in 25 steps at PC = 0x1d.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000c00

Changes to memory:
0x01f8: 0x0000000000000000      0x000000000000001d
```

可以看到寄存器%rax的值为0xcba，正好是三个value相加的结果

rsum.ys

这道题涉及到了递归，那就要对栈进行设计并维护寄存器和栈的值。

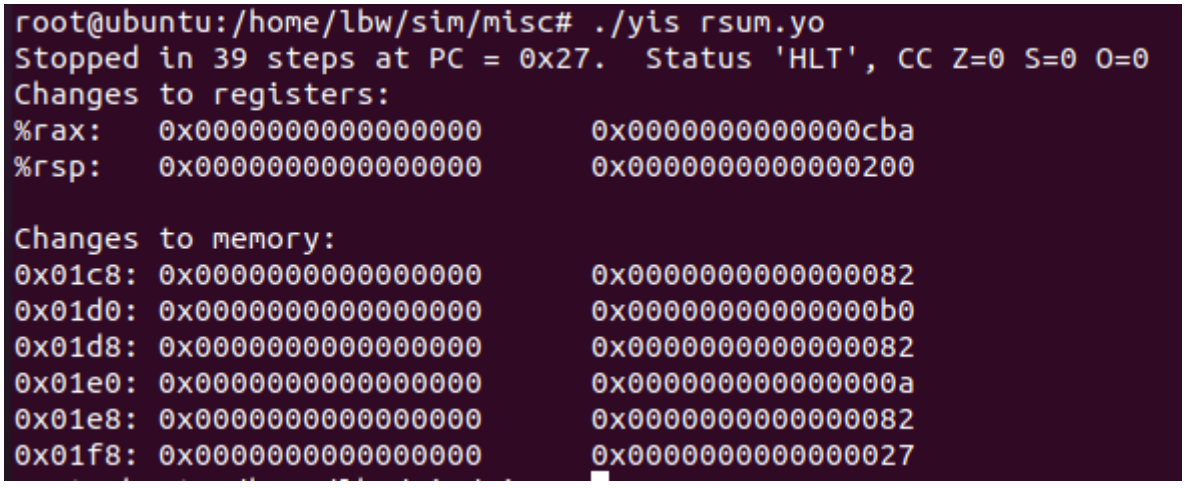
跟上一个函数一样，我们仍然用寄存器%rdi保存参数，寄存器%rax保存返回结果。那么显而易见的，寄存器%rax是我们一定要维护的，因为如果不把它存储到栈中，递归调用的函数会将其的值覆盖；寄存器%rdi也同理，但是有一点不同，%rdi的值在递归函数开始返回之后就再也用不到了，所以我们维护它是没有收益的，所以在这里我选择不维护寄存器%rdi而只维护寄存器%rax

代码如下：

```
# name : Bowen Liu
# ID : 1120201883
# date : 2022.11.12

.pos 0
    irmovq stack,%rsp
    irmovq ele1,%rdi
    irmovq $0,%rax
    call rsum
    halt

.align 8
ele1:
    .quad 0x00a
    .quad ele2
```

```
ele2:
    .quad 0x0b0
    .quad ele3
ele3:
    .quad 0xc00
    .quad 0

rsum:
    andq %rdi,%rdi
    jmp test

calc:
    pushq %rax
    mrmovq (%rdi),%rax
    mrmovq 8(%rdi),%rdi
    call rsum
    popq %r8
    addq %r8,%rax
    ret

test:
    jne calc
    ret

.pos 0x200
stack:
```

运行结果如图：



寄存器%rax的值是0xcba，即三个元素的value之和，程序正确；并且下面的栈变化也可以看出结果的正确性

copy.ys

寄存器%rdi保存参数src，%rsi保存参数dest，%rdx保存参数len，%rax保存返回值

这个函数没什么难的，只要看明白他给的C语言函数要干什么就可以了

代码如下：

```
# name : Bowen Liu
```

```
# ID : 1120201883
# date : 2022.11.12

.pos 0
    irmovq stack, %rsp
    irmovq src,%rdi
    irmovq dest,%rsi
    irmovq $3,%rdx
    call copy
    halt

.align 8
src:
    .quad 0x00a
    .quad 0x0b0
    .quad 0xc00
dest:
    .quad 0x111
    .quad 0x222
    .quad 0x333

copy:
    irmovq $0,%rax
    irmovq $0,%r8
    irmovq $1,%r9
    irmovq $8,%r10
    subq %r8,%rdx
    jmp test

loop:
    mrmovq (%rdi),%r11
    addq %r10,%rdi
    rmmovq %r11,(%rsi)
    addq %r10,%rsi
    xorq %r11,%rax
    subq %r9,%rdx

test:
    jg loop
    ret

.pos 0x200
stack:
```

运行结果如图:

```
root@ubuntu:/home/lbw/sim/misc# ./yis copy.yo
Stopped in 35 steps at PC = 0x31.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000068
%rdi:   0x0000000000000000      0x0000000000000050
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000000000000008
%r11:   0x0000000000000000      0x0000000000000c00

Changes to memory:
0x0050: 0x0000000000000111      0x000000000000000a
0x0058: 0x0000000000000222      0x00000000000000b0
0x0060: 0x0000000000000333      0x0000000000000c00
0x01f8: 0x0000000000000000      0x0000000000000031
```

可以看到原本存放0x111, 0x222, 0x333的内存区域被覆写为了0xa, 0xb0, 0xc00；并且%rax的值为0xcba，说明程序是正确的

Part B

指令跟踪

首先对iaddq指令在SEQ各个阶段做指令跟踪：

阶段	iaddq V , rB
取指	$icode : ifun \leftarrow M_1[PC]$
	$r_A : r_B \leftarrow M_1[PC+1]$
	$valC \leftarrow M_1[PC+2]$
	$valP \leftarrow PC+2$
译码	$valB \leftarrow R[r_B]$
执行	$valE \leftarrow valC + valB$
访存	
写回	$R[r_B] \leftarrow valE$
更新PC	$PC \leftarrow valP$

修改HCL表达式

明确好了各个阶段该干什么，那就去更新各个阶段的hcl表达式

- 取指

  更新instr_valid, need_regids, need_valC，在集合内添加IADDQ
- 译码与写回

更新srcB, dstE

- 执行

  更新aluA, aluB

- 访存

  不需要更改

- 更新PC

  不需要更改

具体的修改:

```
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 };

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
];

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
];
```

测试正确性

修改完毕之后重新构建，并做简单测试：

```
root@ubuntu:/home/lbw/sim/seq# ./ssim -t ../y86-code/asumi.yo
Y86-64 Processor: seq-full.hcl
137 bytes of code read
IF: Fetched irmovq at 0x0.  ra=----, rb=%rsp, valC = 0x100
IF: Fetched call at 0xa.  ra=----, rb=----, valC = 0x38
Wrote 0x13 to address 0xf8
IF: Fetched irmovq at 0x38.  ra=----, rb=%rdi, valC = 0x18
IF: Fetched irmovq at 0x42.  ra=----, rb=%rsi, valC = 0x4
IF: Fetched call at 0x4c.  ra=----, rb=----, valC = 0x56
Wrote 0x55 to address 0xf0
IF: Fetched xorq at 0x56.  ra=%rax, rb=%rax, valC = 0x0
IF: Fetched andq at 0x58.  ra=%rsi, rb=%rsi, valC = 0x0
IF: Fetched jmp at 0x5a.  ra=----, rb=----, valC = 0x83
IF: Fetched jne at 0x83.  ra=----, rb=----, valC = 0x63
IF: Fetched mrmovq at 0x63.  ra=%r10, rb=%rdi, valC = 0x0
IF: Fetched addq at 0x6d.  ra=%r10, rb=%rax, valC = 0x0
IF: Fetched iaddq at 0x6f.  ra=----, rb=%rdi, valC = 0x8
IF: Fetched iaddq at 0x79.  ra=----, rb=%rsi, valC = 0xffffffffffffffff
IF: Fetched jne at 0x83.  ra=----, rb=----, valC = 0x63
IF: Fetched mrmovq at 0x63.  ra=%r10, rb=%rdi, valC = 0x0
IF: Fetched addq at 0x6d.  ra=%r10, rb=%rax, valC = 0x0
IF: Fetched iaddq at 0x6f.  ra=----, rb=%rdi, valC = 0x8
IF: Fetched iaddq at 0x79.  ra=----, rb=%rsi, valC = 0xffffffffffffffff
IF: Fetched jne at 0x83.  ra=----, rb=----, valC = 0x63
IF: Fetched mrmovq at 0x63.  ra=%r10, rb=%rdi, valC = 0x0
IF: Fetched addq at 0x6d.  ra=%r10, rb=%rax, valC = 0x0
IF: Fetched iaddq at 0x6f.  ra=----, rb=%rdi, valC = 0x8
IF: Fetched iaddq at 0x79.  ra=----, rb=%rsi, valC = 0xffffffffffffffff
IF: Fetched jne at 0x83.  ra=----, rb=----, valC = 0x63
IF: Fetched mrmovq at 0x63.  ra=%r10, rb=%rdi, valC = 0x0
IF: Fetched addq at 0x6d.  ra=%r10, rb=%rax, valC = 0x0
IF: Fetched iaddq at 0x6f.  ra=----, rb=%rdi, valC = 0x8
IF: Fetched iaddq at 0x79.  ra=----, rb=%rsi, valC = 0xffffffffffffffff
IF: Fetched jne at 0x83.  ra=----, rb=----, valC = 0x63
IF: Fetched ret at 0x8c.  ra=----, rb=----, valC = 0x0
IF: Fetched ret at 0x55.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x13.  ra=----, rb=----, valC = 0x0
32 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000100
%rdi:   0x0000000000000000      0x0000000000000038
%r10:   0x0000000000000000      0x0000a000a000a000
Changed Memory State:
0x00f0: 0x0000000000000000      0x0000000000000055
0x00f8: 0x0000000000000000      0x0000000000000013
ISA Check Succeeds
```

看到上图最下方的"ISA Check Succeeds"即可得知本次测试通过，之后进行基准程序自动测试：

```
root@ubuntu:/home/lbw/sim/y86-code# make testssim
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t asumr.yo > asumr.seq
../seq/ssim -t cjr.yo > cjr.seq
../seq/ssim -t j-cc.yo > j-cc.seq
../seq/ssim -t poptest.yo > poptest.seq
../seq/ssim -t pushquestion.yo > pushquestion.seq
../seq/ssim -t pushtest.yo > pushtest.seq
../seq/ssim -t prog1.yo > prog1.seq
../seq/ssim -t prog2.yo > prog2.seq
../seq/ssim -t prog3.yo > prog3.seq
../seq/ssim -t prog4.yo > prog4.seq
../seq/ssim -t prog5.yo > prog5.seq
../seq/ssim -t prog6.yo > prog6.seq
../seq/ssim -t prog7.yo > prog7.seq
../seq/ssim -t prog8.yo > prog8.seq
../seq/ssim -t ret-hazard.yo > ret-hazard.seq
grep "ISA Check" *.seq
asum.seq:ISA Check Succeeds
asumr.seq:ISA Check Succeeds
cjr.seq:ISA Check Succeeds
j-cc.seq:ISA Check Succeeds
poptest.seq:ISA Check Succeeds
prog1.seq:ISA Check Succeeds
prog2.seq:ISA Check Succeeds
prog3.seq:ISA Check Succeeds
prog4.seq:ISA Check Succeeds
prog5.seq:ISA Check Succeeds
prog6.seq:ISA Check Succeeds
prog7.seq:ISA Check Succeeds
prog8.seq:ISA Check Succeeds
pushquestion.seq:ISA Check Succeeds
pushtest.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asum.seq asumr.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushtest.seq prog1.seq prog2.se
q prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq prog8.seq ret-hazard.seq
```

全部通过，那么接下来进行大量回归测试：

```
root@ubuntu:/home/lbw/sim/ptest# make SIM=../seq/ssim
./optest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 49 ISA Checks Succeed
./jtest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 600 ISA Checks Succeed
```

非iaddq指令全部通过测试

```
root@ubuntu:/home/lbw/sim/ptest# make SIM=../seq/ssim TFLAGS=-i
./optest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 58 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 756 ISA Checks Succeed
```

包含iaddq指令的也全部通过测试

至此partB全部完成

Part C

这个部分没有特别明确的实验完成的要求，但是想要得到满分需要将CPE降至7.5以下。读完实验文档的描述后，欧我能想到的降低CPE的方法有：

- 减少指令数

  原程序中先把常数赋给寄存器，然后用寄存器去做opq操作；若能直接做iopq操作则能节省irmovq指令

- 使用条件跳转指令尽可能令它跳转

  Pipe的分支预测策略是“总是跳转”，每当预测错误时就会损失两个时钟周期的时间

- 避免加载/使用冒险

原本的ncopy.ys:

```
ncopy:
    xorq %rax,%rax              # count = 0;
    andq %rdx,%rdx              # len <= 0?
    jle Done                    # if so, goto Done:

Loop:
    mrmovq (%rdi), %r10         # read val from src...
    rmmovq %r10, (%rsi)         # ...and store it to dst
    andq %r10, %r10             # val <= 0?
    jle Npos                    # if so, goto Npos:
    irmovq $1, %r10
    addq %r10, %rax             # count++

Npos:
    irmovq $1, %r10
    subq %r10, %rdx             # len--
    irmovq $8, %r10
    addq %r10, %rdi             # src++
    addq %r10, %rsi             # dst++
    andq %rdx,%rdx              # len > 0?
    jg Loop                     # if so, goto Loop:
```

为Pipe添加iaddq指令

想要减少指令数，我们需要先修改pipe-full.hcl以实现流水线化的iaddq指令。在SEQ中添加iaddq指令在partB已经实现过了，这里只会说明一下与SEQ不同的地方：

1. 取指

   与SEQ相同

2. 译码与写回

   Pipe中的转发逻辑已经写好了，所以这一阶段需要修改的与SEQ也相同

3. 执行

set_CC的更改略有不同

4. 访存

   与SEQ相同。这一阶段不需要修改

5. 控制逻辑

   首先指令iaddq是不会引起控制冒险的，它会引起的只有数据冒险；数据冒险中要在控制逻辑中有体现的是加载/控制冒险，而iaddq指令也同样不会触发控制冒险；其他种类的数据冒险用转发即可解决

```
bool set_cc = (E_icode in {IOPQ, IIADDQ}) &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
```

验证添加iaddq指令的Pipe的正确性

基准程序测试：

```
root@ubuntu:/home/lbw/sim/pipe# (cd ../y86-code; make testpsim)
../pipe/psim -t asum.yo > asum.pipe
../pipe/psim -t asumr.yo > asumr.pipe
../pipe/psim -t cjr.yo > cjr.pipe
../pipe/psim -t j-cc.yo > j-cc.pipe
../pipe/psim -t poptest.yo > poptest.pipe
../pipe/psim -t pushquestion.yo > pushquestion.pipe
../pipe/psim -t pushtest.yo > pushtest.pipe
../pipe/psim -t prog1.yo > prog1.pipe
../pipe/psim -t prog2.yo > prog2.pipe
../pipe/psim -t prog3.yo > prog3.pipe
../pipe/psim -t prog4.yo > prog4.pipe
../pipe/psim -t prog5.yo > prog5.pipe
../pipe/psim -t prog6.yo > prog6.pipe
../pipe/psim -t prog7.yo > prog7.pipe
../pipe/psim -t prog8.yo > prog8.pipe
../pipe/psim -t ret-hazard.yo > ret-hazard.pipe
grep "ISA Check" *.pipe
asum.pipe:ISA Check Succeeds
asumr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
j-cc.pipe:ISA Check Succeeds
poptest.pipe:ISA Check Succeeds
prog1.pipe:ISA Check Succeeds
prog2.pipe:ISA Check Succeeds
prog3.pipe:ISA Check Succeeds
prog4.pipe:ISA Check Succeeds
prog5.pipe:ISA Check Succeeds
prog6.pipe:ISA Check Succeeds
prog7.pipe:ISA Check Succeeds
prog8.pipe:ISA Check Succeeds
pushquestion.pipe:ISA Check Succeeds
pushtest.pipe:ISA Check Succeeds
ret-hazard.pipe:ISA Check Succeeds
rm asum.pipe asumr.pipe cjr.pipe j-cc.pipe poptest.pipe pushques
pipe prog3.pipe prog4.pipe prog5.pipe prog6.pipe prog7.pipe prog
```

回归测试：

```
root@ubuntu:/home/lbw/sim/pipe#  (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 58 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 756 ISA Checks Succeed
```

测试全部成功通过

下面对这个汇编程序可以优化的地方进行分析：

优化

条件跳转指令优化

1. 第3条指令 `jle Done` 会跳转到ret指令，但是一个while循环只有最后一次会跳转，其他时间下是不会跳转的。把它修改为 `jg Loop` 并将其放在Done之前会好很多
2. 第7条指令 `jle Npos` 没什么优化空间
3. 最后一条指令 `jg Loop` 可以与1中所说的那个指令合并

减少指令数

有了iaddq指令之后就可以把常数0与常数8的赋值指令去掉了，汇编代码：

```
# You can modify this portion
    # Loop header
    xorq %rax,%rax
    andq %rdx,%rdx
    jmp Test

Loop:
    mrmovq (%rdi), %r10
    rmmovq %r10, (%rsi)
    andq %r10, %r10
    jle Npos
    iaddq $1,%rax
Npos:
    irmovq $1, %r10
    subq %r10, %rdx
    iaddq $8, %rdi
    iaddq $8, %rsi
    andq %rdx,%rdx

Test:
    jg Loop
```

操作之后对程序进行测试：

```
Average CPE     13.62
Score   0.0/60.0
```

看到这里我心态崩了，本以为做出这些优化可以提升不少性能了，但是提升了，但只提升了一点点，甚至1分都没拿到。这时我就在想：是不是我优化的方向出了问题？

所以我就重新去看实验的文档，发现作者有提示去看5.8关于循环展开的内容，那可能这个程序的优化就要从这一小节中的内容入手了

循环展开

为了对循环展开对程序优化的能力做出一个估计，我先写了一个4×1循环展开程序，如下：

```
xorq %rax,%rax
iaddq $-4,%rdx
jl Root

Loop1:
    mrmovq (%rdi), %r8
    mrmovq 8(%rdi), %r9
    rmmovq %r8, (%rsi)
    andq %r8, %r8
    jle Loop2
    iaddq $1, %rax

Loop2:
    mrmovq 16(%rdi), %r8
    rmmovq %r9, 8(%rsi)
    andq %r9, %r9
    jle Loop3
    iaddq $1, %rax

Loop3:
    mrmovq 24(%rdi), %r9
    rmmovq %r8, 16(%rsi)
    andq %r8, %r8
    jle Loop4
    iaddq $1, %rax

Loop4:
    rmmovq %r9, 24(%rsi)
    andq %r9, %r9
    jle Loop
    iaddq $1, %rax

Loop:
    iaddq $32,%rdi
    iaddq $32,%rsi
    iaddq $-4,%rdx
    jge Loop1

# 余数处理，直接进行循环
Test:
    iaddq $4,%rdx
    je Done

NLoop:
    mrmovq (%rdi), %r8
    rmmovq %r8, (%rsi)
```

```
    andq %r8, %r8
    jle Npos
    iaddq $1,%rax

Npos:
    irmovq $1, %r8
    subq %r8, %rdx
    iaddq $8, %rdi
    iaddq $8, %rsi
    andq %rdx,%rdx
    jg NLoop
```

直接对其进行benchmark测试：

```
Average CPE     8.63
Score   37.4/60.0
```

发现仅仅是4×1的循环展开就已经对程序与得分产生了巨大的影响，我就将循环展开扩大到8×1，然后再进行benchmark，如图：

```
Average CPE     8.87
Score   32.6/60.0
```

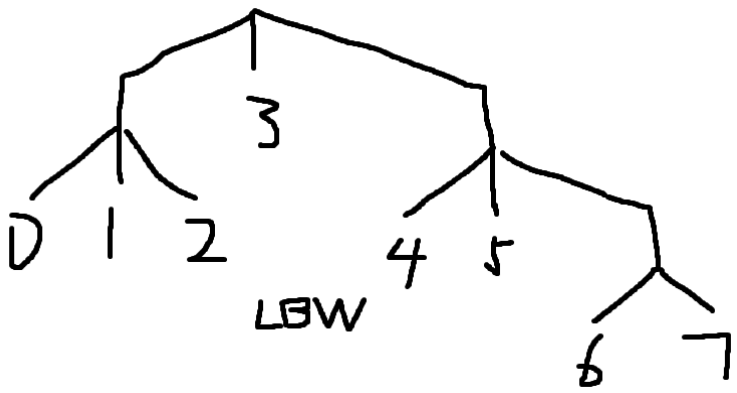
我发现8×1的循环展开竟然还不如4×1！我查看具体复制块数所对应时间尝试分析原因，当展开因子提升时，理论上来说循环的效率的是要提升的；但是影响程序效率的还有另外一个因素：`在循环上限以外的那些余数`。这些余数的处理是直接循环，这相比循环展开是要慢非常多的；体现在benchmark里可以看到当循环8×1展开时，复制块数为8的倍数的CPE很低，而复制块数为8的倍数-1的就要高很多，如图：

```
8       67      8.38
9       82      9.11
10      96      9.60
11      107     9.73
12      121     10.08
13      132     10.15
14      146     10.43
15      157     10.47
16      115     7.19
```

那么下一步就去优化余数

余数优化

这里要对余数做一些优化，也就是对剩余的循环做优化。这里我想了好久没有思路，就去互联网上搜寻解决方案。看到一个大佬，使用`三叉搜索树`来对余数的每一种情况进行优化：汇编指令做一次opq指令会设置条件码，jl，jg，je会分出三种情况，对应三叉搜索数的三个分支。而对于8×1的循环展开，我画了一下它的搜索树如图：



确定了余数是几之后，就可以进行针对性的优化，以顺序的程序结构替代循环的跳转指令。这样在余数的运算中就去除了所有的循环跳转指令，类似循环展开

最终结果

汇编代码：

```
xorq %rax,%rax
iaddq $-8,%rdx
jl Test

Loop1:
    mrmovq (%rdi), %r8
    mrmovq 8(%rdi), %r9
    rmmovq %r8, (%rsi)
    andq %r8, %r8
    jle Loop2
    iaddq $1, %rax

Loop2:
    mrmovq 16(%rdi), %r8
    rmmovq %r9, 8(%rsi)
    andq %r9, %r9
    jle Loop3
    iaddq $1, %rax

Loop3:
    mrmovq 24(%rdi), %r9
    rmmovq %r8, 16(%rsi)
    andq %r8, %r8
    jle Loop4
    iaddq $1, %rax

Loop4:
    mrmovq 32(%rdi), %r8
    rmmovq %r9, 24(%rsi)
    andq %r9, %r9
    jle Loop5
    iaddq $1, %rax

Loop5:
    mrmovq 40(%rdi), %r9
    rmmovq %r8, 32(%rsi)
    andq %r8, %r8
    jle Loop6
    iaddq $1, %rax

Loop6:
    mrmovq 48(%rdi), %r8
    rmmovq %r9, 40(%rsi)
    andq %r9, %r9
    jle Loop7
    iaddq $1, %rax

Loop7:
    mrmovq 56(%rdi), %r9
    rmmovq %r8, 48(%rsi)
    andq %r8, %r8
    jle Loop8
```

```
    iaddq $1, %rax

Loop8:
    rmmovq %r9, 56(%rsi)
    andq %r9, %r9
    jle Loop
    iaddq $1, %rax

Loop:
    iaddq $64,%rdi
    iaddq $64,%rsi
    iaddq $-8,%rdx
    jge Loop1

Test:
    iaddq $5,%rdx
    jg RightNode1
    jl LeftNode
    je Remain3

RightNode1:
    iaddq $-2,%rdx
    jg RightNode2
    jl Remain4
    je Remain5

RightNode2:
    iaddq $-1,%rdx
    jg Remain7
    je Remain6

LeftNode:
    iaddq $2,%rdx
    je Remain1
    jl Done
    iaddq $-1,%rdx
    jmp Remain2

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
```

最终采取8 × 1加余数优化的benchmark测试结果为：

```
Average CPE     7.65
Score   57.0/60.0
```

正确性测试：

```
68/68 pass correctness test
```

一点感想

很遗憾努力了这么久也最终没能得到60分，但是在这个实验的partC中我学到了很多优化相关的知识，也算是为第五章的学习做准备把