Part A

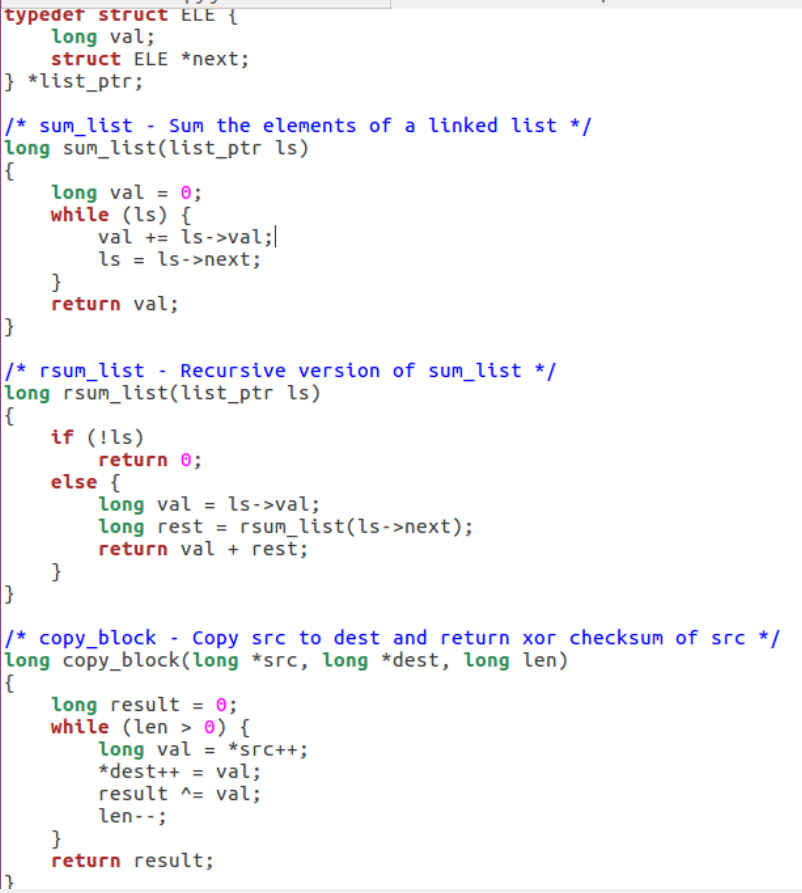
sum.ys对链表求和

rsum.ys递归地对链表求和, 具体代码和之前的函数差不多，需要注意的是因为有**递归**，所以需要用**栈帧**（stack frame)**保存递归实例信息**，所以需要**保存寄存器信息并还原**。我在这里使用了被调用者保存寄存器(callee-save )**%r12,** 所以在函数的**一开始保存寄存器信息**，**在ret前恢复**。

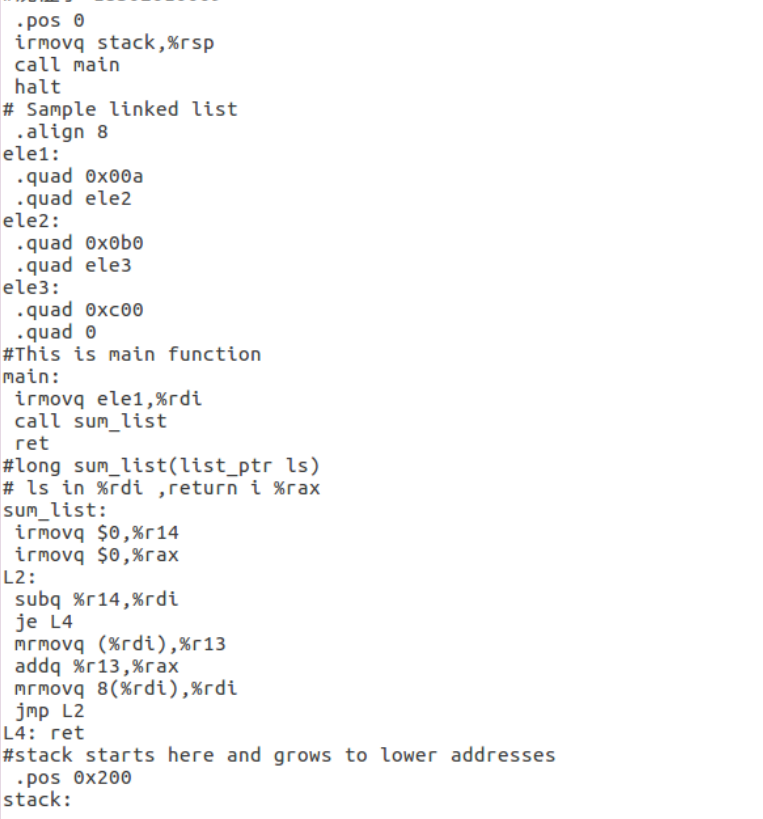
and copy.ys. 数组复制 这里要注意，因为y86-64 **只支持寄存器的运算操作**，所以需要将立即数1 8 存入寄存器中。当然也不要忘记了**先保存再还原值**。

编写Y86-64程序

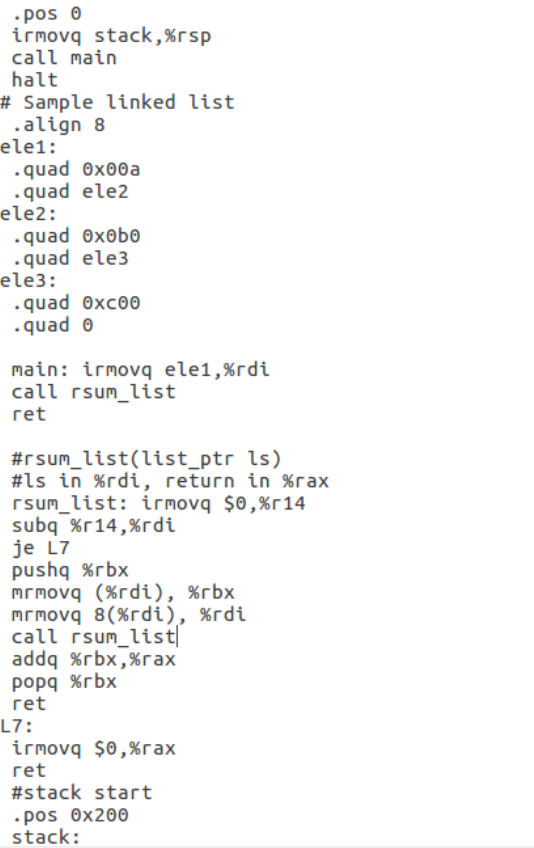
例程：



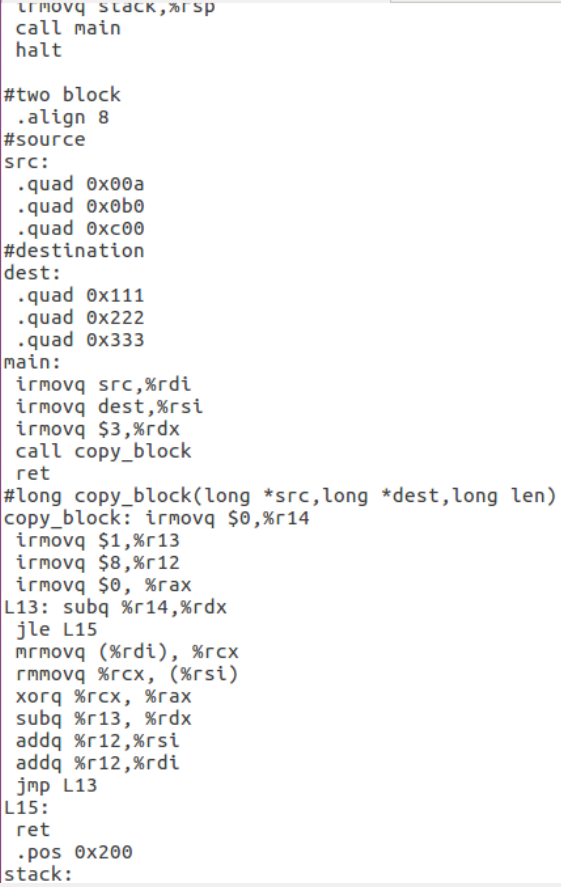
Sum.ys



Rsum.ys



Copy.ys

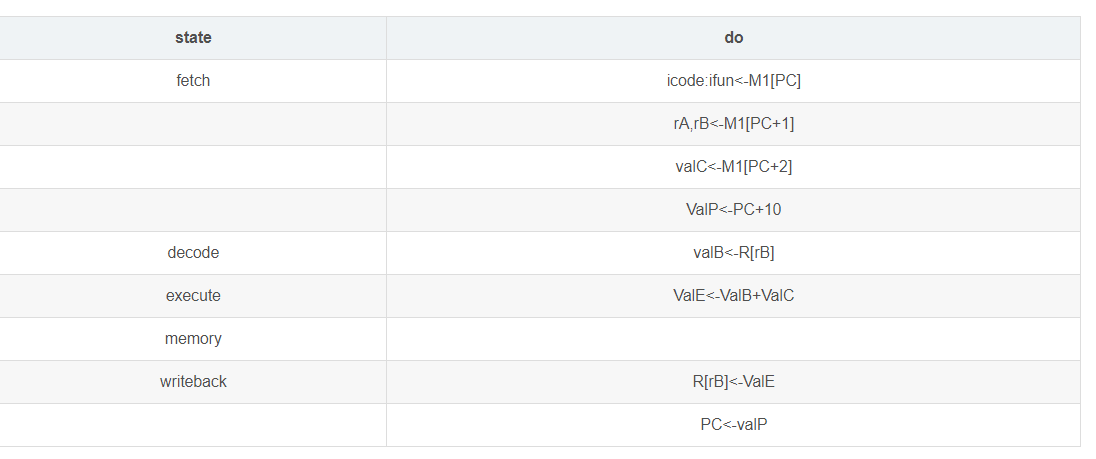


Part B:

给SEQ处理器添加iaddq指令

iaddq指令描述

修改HCL代码就照着改，大概就是有OPQ的地方都加一个IIADDQ，注意参与运算的值不是valA而是valC



# The five steps of iaddq:

# fetch: icode:ifun = M1[PC] rA:rB = M1[PC] valC = M8[PC+2] valP = PC + 10

# decode: valB = R[rB]

# execute: valE = valC + valB; set CC

# memory:

# write back: R[rB] = valE

# PC updated: PC = valP

先要知道iaddq每步都做什么

valA和valB，只要valB就够了（获得哪个寄存器）

valC是常数值

decode阶段只要去拿rB寄存器的值就够了

execute阶段将valB和valC相加，赋值给valE，再写回rB寄存器

################ Fetch Stage ###################################

# Determine instruction code

word icode = [

imem\_error: INOP;

1: imem\_icode; # Default: get from instruction memory

];

# Determine instruction function

word ifun = [

imem\_error: FNONE;

1: imem\_ifun; # Default: get from instruction memory

];

bool instr\_valid = icode in

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

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

# Does fetched instruction require a regid byte?

bool need\_regids =

icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIADDQ,

IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?

bool need\_valC =

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

################ Decode Stage ###################################

## What register should be used as the A source?

word srcA = [

icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;

icode in { IPOPQ, IRET } : RRSP;

1 : RNONE; # Don't need register

];

## What register should be used as the B source?

word srcB = [

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

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

1 : RNONE; # Don't need register

];

## What register should be used as the E destination?

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

];

## What register should be used as the M destination?

word dstM = [

icode in { IMRMOVQ, IPOPQ } : rA;

1 : RNONE; # Don't write any register

];

################ Execute Stage ###################################

## Select input A to ALU

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

];

## Select input B to 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

];

## Set the ALU function

word alufun = [

icode == IOPQ : ifun;

1 : ALUADD;

];

## Should the condition codes be updated?

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

################ Memory Stage ###################################

## Set read control signal

bool mem\_read = icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal

bool mem\_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

## Select memory address

word mem\_addr = [

icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;

icode in { IPOPQ, IRET } : valA;

# Other instructions don't need address

];

## Select memory input data

word mem\_data = [

# Value from register

icode in { IRMMOVQ, IPUSHQ } : valA;

# Return PC

icode == ICALL : valP;

# Default: Don't write anything

];

## Determine instruction status

word Stat = [

imem\_error || dmem\_error : SADR;

!instr\_valid: SINS;

icode == IHALT : SHLT;

1 : SAOK;

];

################ Program Counter Update ############################

## What address should instruction be fetched at

word new\_pc = [

# Call. Use instruction constant

icode == ICALL : valC;

# Taken branch. Use instruction constant

icode == IJXX && Cnd : valC;

# Completion of RET instruction. Use value from stack

icode == IRET : valM;

# Default: Use incremented PC

1 : valP;

];

#/\* $end seq-all-hcl \*/

Part C：

优化程序性能

修改PIPE 的HCL描述，以及给定的一个函数的y86-64 代码，提高CPE。

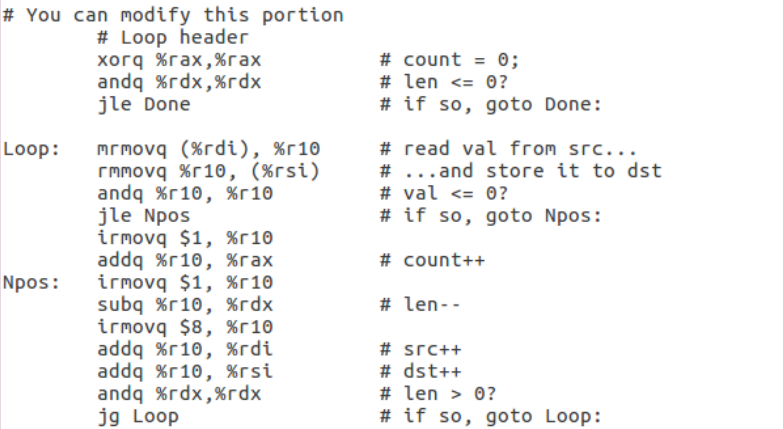
首先是修改pipe目录下的pipe-full.hcl 文件，增加iaddq指令。

在tty自动测试、基准程序测试、回归测试全部通过后，我们再来修改Y86-64 程序。

将len element整数数组src复制到非重叠dst，返回src中包含的正整数的计数。

我们首先写出不用循环展开，能减小CPE的程序，修改了如下部分：

原来的版本



* 消除load/use hazzard
* 根据pipe-full的条件跳转预测实现，修改jmp指令位置，减少预测错误损失
* 因为y86-64没有cmpq命令，用巧妙的方法进行条件码设置
* 循环展开

