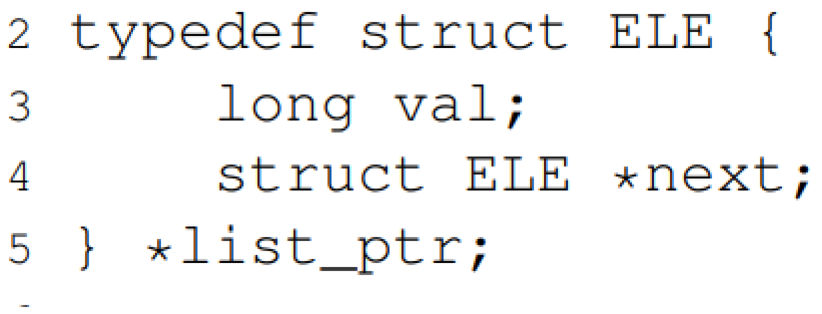
# Part A

`工作目录: sim/misc`

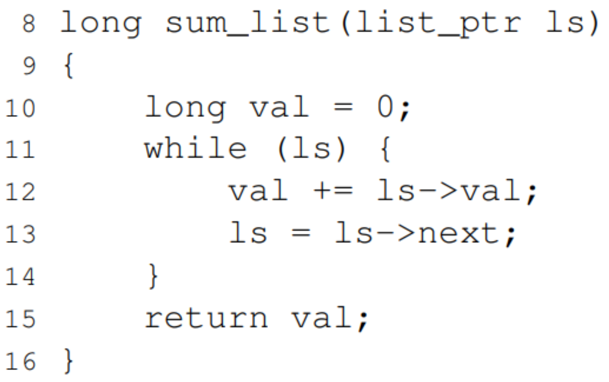
本任务要求我们把三个`C` 语言函数：`sum_list`, `rsum_list`, `copy_block`翻译成`Y86` 汇编代码， 这三个函数的源码放在了文件`example.c` .

其定义的节点结构体形如:



```
typedef struct ELE {
      long val;
      struct ELE *next;
} *list_ptr;
```

## sum_list



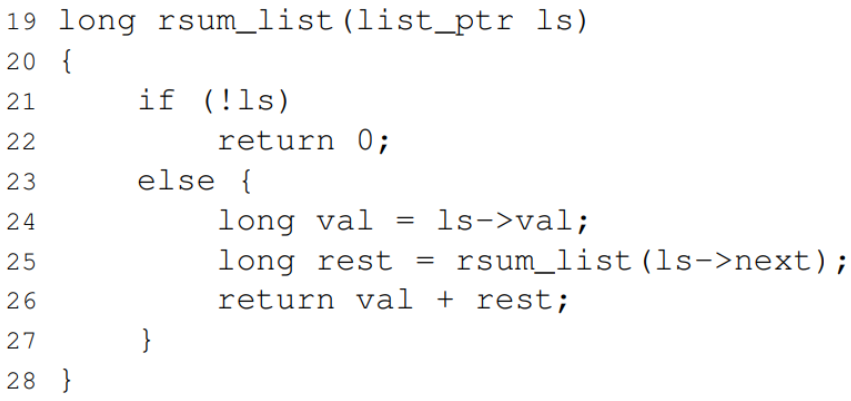
```
long sum_list(list_ptr ls)
{
    long val = 0;
    while (ls) {
        val += ls->val;
        ls = ls->next;
    }
    return val;
}
```

我们发现这就是个基本的按照顺序求和链表所有节点的值。

这里建议先翻一下书，不然就会像我一样，连一个完整的汇编代码的格式都不清楚。

```
# sum_list
# Execution begins at address 0
        .pos 0
        irmovq stack %rsp
        call main
        halt
# sample linked list
        .align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
main:
    irmovq ele1,%rdi # 把首节点的地址放入第一个参数寄存器
    call sum_list
    res

sum_list:
        irmovq $0, %rax # long val = 0;
        andq %rdi, %rdi # set CC
        jmp test
loop:
        mrmovq (%rdi), %rsi # ls->val
        addq %rsi, %rax # val+=ls->val
        mrmovq 8(%rdi), %rdi # ls = ls->next


test:
        jne loop # if (ls!=0)
        ret

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:

```

## rsum_list

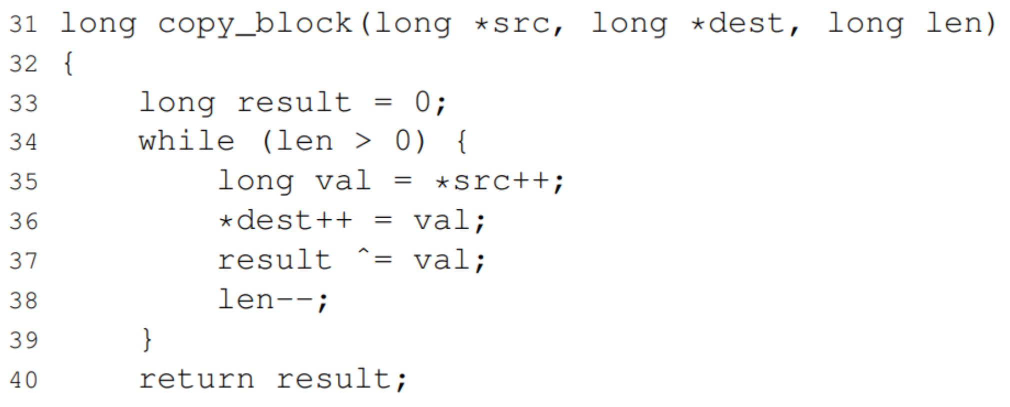


```
long rsum_list(list_ptr ls)
{
    if (!ls)
        return 0;
    else {
        long val = ls->val;
        long rest = rsum_list(ls->next);
        return val + rest;
    }
}
```

很明显这是一个递归过程。

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp      # Set up stack pointer
        call main               # Execute main program
        halt                    # Terminate program


# Sample linked list
        .align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0

main:
        irmovq ele1,%rdi
        call rsum_list
        ret


rsum_list:
        andq %rdi, %rdi
        je return  if (!ls)
        mrmovq (%rdi), %rbx # val = ls->val
        mrmovq 8(%rdi), %rdi # ls = ls->next
        pushq %rbx
        call rsum_list
        pushq %rbx  # 保存%rbx
        call rsum_list
        popq %rbx
        addq %rbx, %rax
        ret

return:
        irmovq $0, %rax
        ret

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:

```

# copy_block



```
long copy_block(long *src, long *dest, long len)
{
    long result = 0;
    while (len > 0) {
        long val = *src++;
        *dest++ = val;
        result ^= val;
        len--;
    }
    return result;
```

```
# Execution begins at address 0
```

```
        .pos 0
        irmovq stack, %rsp      # Set up stack pointer
        call main               # Execute main program
        halt                    # Terminate program

# Sample
        .align 8
# Source block
src:
        .quad 0x00a
        .quad 0x0b0
        .quad 0xc00

# Destination block
dest:
        .quad 0x111
        .quad 0x222
        .quad 0x333

main:
        irmovq src, %rdi        # 把src放入第一个参数寄存器
        irmovq dest, %rsi       # 把dest放入第二个参数寄存器
        irmovq $3, %rdx         # 把len放入第三个参数寄存器
        call copy_block
        ret
copy_block:
        irmovq $8,%r8 #src++ 和 dest++ 所需的立即数
        irmovq $1, %r9 #len--所需的立即数
        irmovq $0 %rax # result = 0
        andq %rdx, %rdx
        jmp test

loop:
        mrmovq (%rdi), %r10 # val = *src
        addq %r8, %rdi # src+=1
        rmmovq %r10, (%rsi) # *dest = val
        addq %r8, %rsi # dest+=1
        xorq %r10, %rax # result^=val
        subq %r9, %rdx # len-=1




test:
        jne loop # if len!=0
        ret

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:
```

# Part B

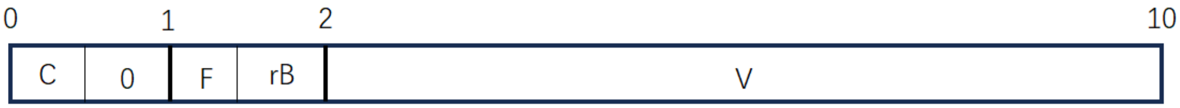
`工作目录: sim/seq`

**实验要求：**

> Your task in Part B is to extend the SEQ processor to support the iaddq, described in Homework problems 4.51 and 4.52. To add this instructions, you will modify the file seq-full.hcl, which implements the version of SEQ described in the CS:APP3e textbook. In addition, it contains declarations of some constants that you will need for your solution.

也就是说：现在我们已经有了一个 `HCL` 文件，里面已经实现了一些指令。现在我们需要考虑新加入一个 `iaddq` 指令，并修改我们的 `HCL` 文件，使得其支持这条指令。

`iaddq V, rB` 的二进制编码长这个样子:



我们现在看看 `iaddq` 在 `pipeline` 的每个 `stage` 是怎么执行的:

- Fetch: icode : ifun ← $M_1$ [PC]

  rA : rB ← $M_1$ [PC+1]

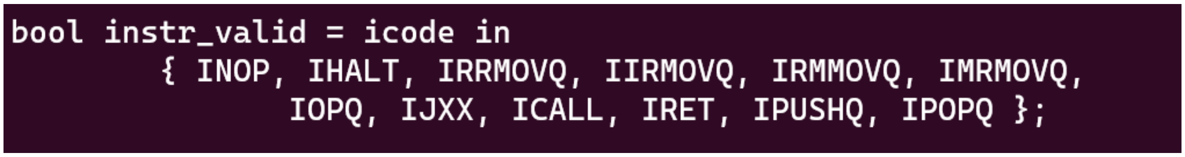
  valC ← $M_8$ [PC+2]

  valP ← PC+10
- Decode: valB ← R[rB]
- Excute： valE ← valB + valC

  set CC
- Memory :
- Write back : R[rB] ← valE
- PC update : PC ← valP

现在我们进入 `seq-full.hcl`, 一条信号一条地分析:

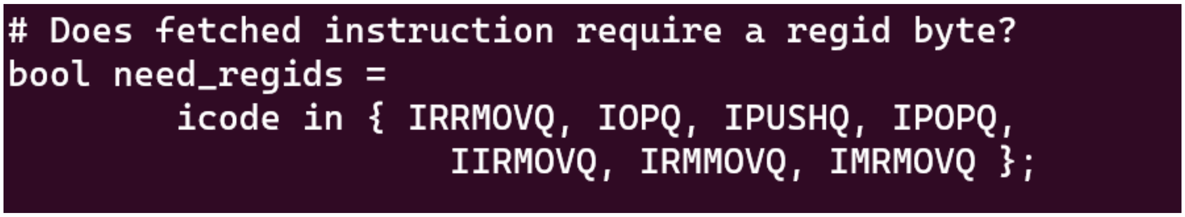
## Fetch:

- 

```
bool instr_valid = icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
               IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```

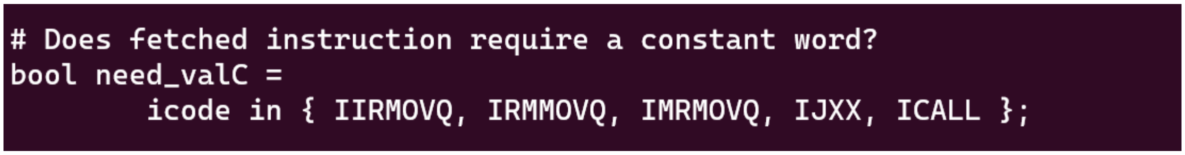
把 `IIADDQ` 加到最后。

-



```
# Does fetched instruction require a regid byte?
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

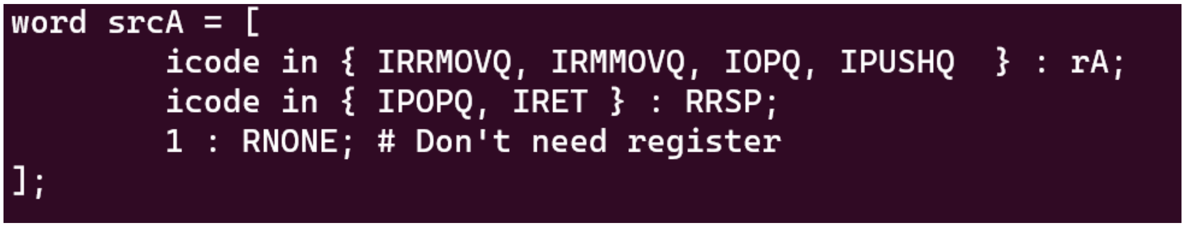
把 `IIADDQ` 加到最后。

- 

```
# Does fetched instruction require a constant word?
bool need_valC =
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
```

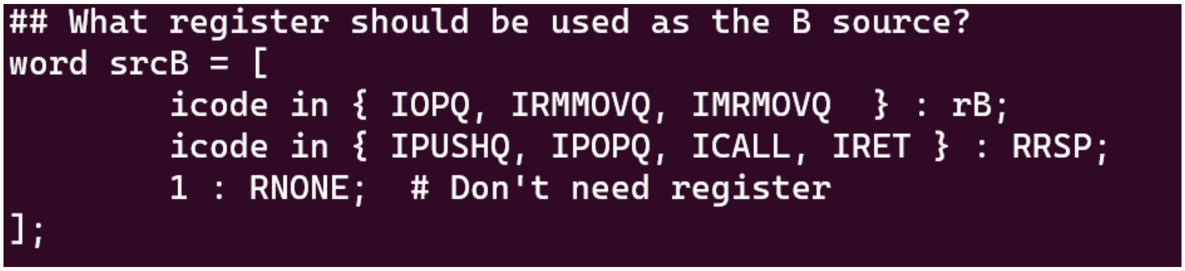
把 `IIADDQ` 加到最后。

## Decode

- 

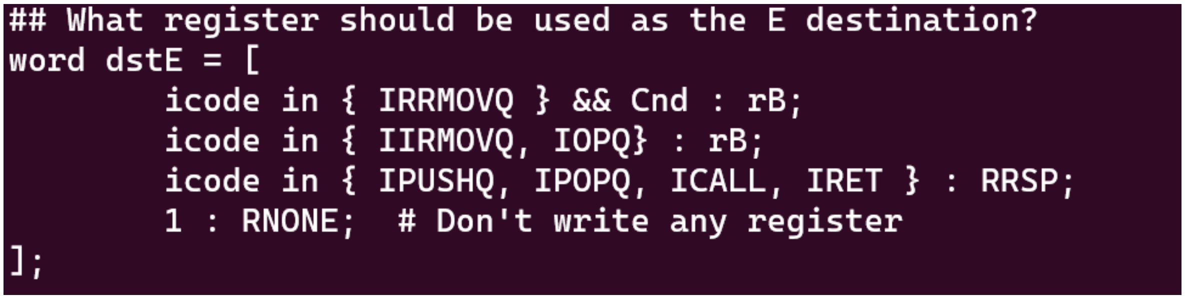
```
word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

不用修改。

- 

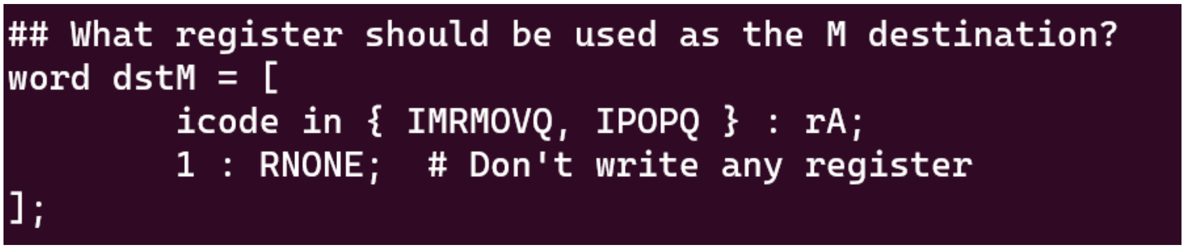
```
## What register should be used as the B source?
word srcB = [
        icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't need register
];
```

在 `rB` 那一行加上 `IIADDQ` 。

- 

```
## What register should be used as the E destination?
word dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

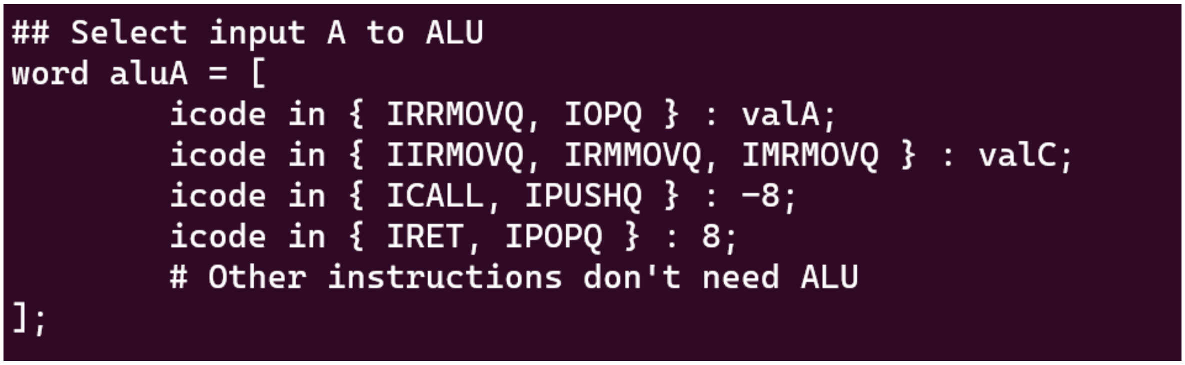
在第二行加上`IIADDQ`.

- 

```
## What register should be used as the M destination?
word dstM = [
        icode in { IMRMOVQ, IPOPQ } : rA;
        1 : RNONE;  # Don't write any register
];
```

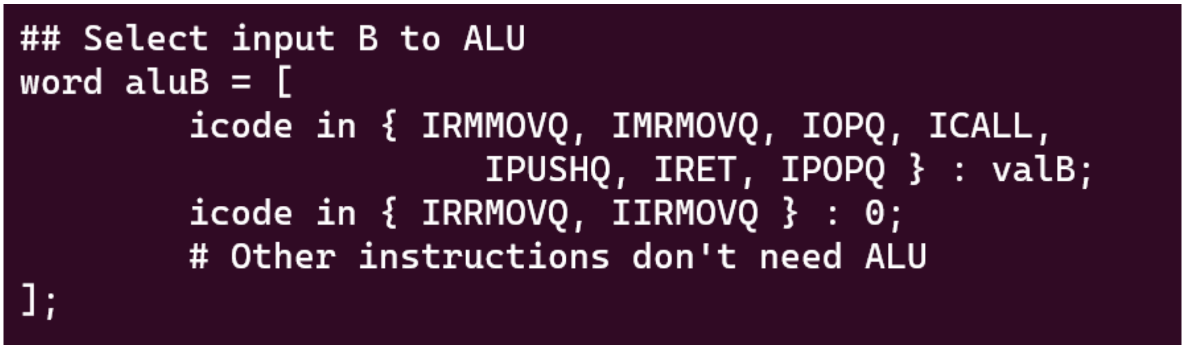
不做修改

## Excute

- 

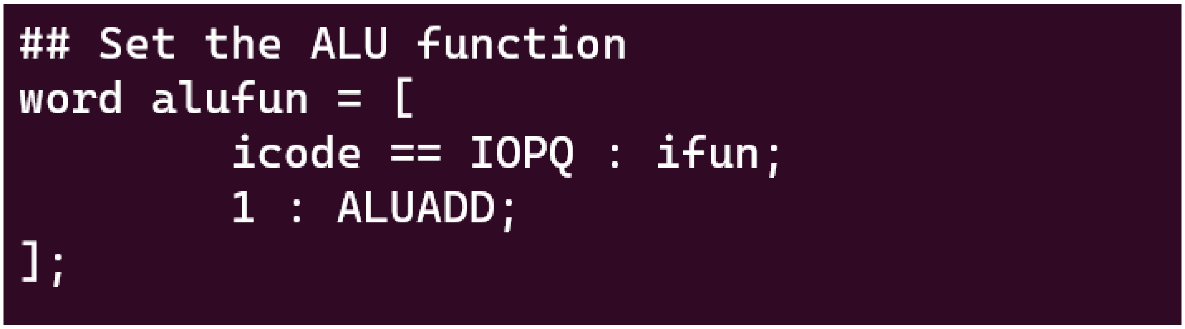
```
## Select input A to ALU
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

`valC` 那行加上`IIADDQ` 。

- 

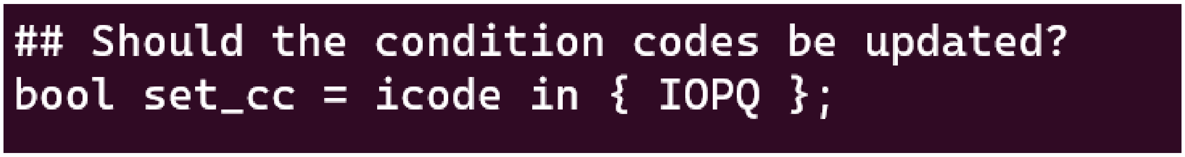
```
## Select input B to ALU
word aluB = [
        icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                      IPUSHQ, IRET, IPOPQ } : valB;
        icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

`valB` 那行加上`IIADDQ`。

- 

```
## Set the ALU function
word alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

不变，也没法变

- 

```
## Should the condition codes be updated?
bool set_cc = icode in { IOPQ };
```

加上`IIADDQ`。

## Memory

都不用修改，根本根内存没关系。

## PC update

next pc 就是默认的valp, 所以也不用管。

# Part C

工作目录：`sim/pipe`

`实验任务`：我们现在有两个文件 `ncopy.c`, `ncopy.ys`, 其中后者是前者的汇编。

`ncopy.c`:

```c
#include <stdio.h>

typedef word_t word_t;

word_t src[8], dst[8];

/* $begin ncopy */
/*
 * ncopy - copy src to dst, returning number of positive ints
 * contained in src array.
 */
word_t ncopy(word_t *src, word_t *dst, word_t len)
{
    word_t count = 0;
    word_t val;

    while (len > 0) {
        val = *src++;
        *dst++ = val;
        if (val > 0)
            count++;
        len--;
    }
    return count;
}
/* $end ncopy */

int main()
{
    word_t i, count;

    for (i=0; i<8; i++)
        src[i]= i+1;
    count = ncopy(src, dst, 8);
    printf ("count=%d\n", count);
    exit(0);
}
```

`ncopy.ys`:

```
#/* $begin ncopy-ys */
##################################################################
# ncopy.ys - Copy a src block of len words to dst.
# Return the number of positive words (>0) contained in src.
#
# Include your name and ID here.
#
# Describe how and why you modified the baseline code.
#
##################################################################
# Do not modify this portion
# Function prologue.
# %rdi = src, %rsi = dst, %rdx = len
ncopy:

##################################################################
# You can modify this portion
        # Loop header
        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:
##################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
        ret
##################################################################
# Keep the following label at the end of your function
End:
#/* $end ncopy-ys */
```

除此之外，我们还有一份`pipe-full.hcl`文件(已经加入了`iaddq`)，是描述我们此时所在的流水线架构`PIPE`的。

现在 我们可以修改`ncopy.ys`和`pipe-full.hcl`, 来使得`ncopy.ys`运行得更快

# 优化开始

- 由于我们的`PIPE`支持`iaddq`, 那么我们可以把所有`++,--`改为使用`iaddq`, 这样可以减少指令数。

```
# You can modify this portion
    # Loop header
    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:
    iaddq $1, %rax      # count++
Npos:
    iaddq $-1, %rdx     # len--
    iaddq $8, %rdi      # src++
    iaddq $8, %rsi      # dst++
    andq %rdx,%rdx      # len > 0?
    jg Loop         # if so, goto Loop:
```