

实验四-优化Y86-64流水线处理器性能

Y86-64指令集编码规范

首先是一些汇编的要求

- 参数的传法，第 1 个参数：%rdi，第 2 个参数：%rsi，第 3 个参数：%rdx，第 4 个参数：%rcx，第 5 个参数：%r8，第 6 个参数：%r9，返回值：%rax。
- 被调用者保存的寄存器：%rbx，%rbp，%r12，%r13，%r14，%r15。

第一段：代码段（Code）

Y86 程序必须从 `.pos 0` 开始编写代码段，这是因为 Y86 CPU 启动时程序计数器 PC 固定为 0，第一条指令一定从内存地址 0 取指执行。`.pos 0` 并不是 CPU 指令，而是汇编器伪指令，用来告诉汇编器把接下来的机器码放在内存地址 0，从而保证 PC 能正确取到第一条指令。所有会被 CPU 执行的内容（主程序、函数代码）都属于代码段。

第二段：数据段（Data）

数据段用于存放程序运行过程中需要访问的静态数据，例如链表节点等，通常通过 `.quad` 指令定义。`.quad` 表示在当前位置放置一个 64 位数据，而标签（如 `ele1:`）只是该地址的名字，本身不占内存。数据段应当放在代码段之后，并通过 `.align 8` 进行 8 字节对齐，以保证数据访问规范且不被 CPU 当作指令执行。

第三段：栈段（Stack）

栈段是程序运行时使用的内存区域，用于保存函数返回地址和局部数据，由寄存器 `%rsp` 管理，且栈向低地址方向增长。通常通过 `.pos 0x200` 将栈起始地址放在较高的内存位置，再用标签 `stack:` 标记该地址。选择 0x200 是一种安全、保守的做法，它远离代码段和数据段，同时小于 Y86 地址空间上限（0xFFFF），并满足 8 字节对齐要求。

第四段：书写顺序与规则总结

在 Y86 实验中，推荐的书写顺序为：代码段 → 数据段 → 栈段。该顺序并非语法强制，而是为了保证程序入口明确、数据不被误执行、栈不覆盖代码和数据。 `.pos 0` 必须保留作为程序入口，数据段只用于存放数据而不执行，栈段必须位于代码和数据之上并预留向下增长空间。而且最后的一行必须有回辙。

Part A

sum.js-对链表元素进行迭代求和

这里就是简单地练习编写Y86代码，下面给出代码sum.js

```
#-----
# sum.js
# Name:徐文博
# 学号:1320240207
# 链表元素求和
#-----

        .pos 0
        irmovq stack, %rsp      # 初始化栈指针
        irmovq ele1, %rdi       # 传入第一个参数，链表头第一个元素地址
        call sum_list           # 调用sum_list函数
        halt                    # 程序结束

#-----
# long sum_list(list_ptr ls)
# %rdi = ls
# 返回指针在%rax中
#-----
sum_list:
        xorq %rax, %rax        #作为返回结果

Loop:
        andq %rdi, %rdi        # 判断参数ls是否为NULL
        je Done                # 如果是NULL，跳转到Done
        mrmovq 0(%rdi), %r10    # 取出当前节点的8字节值并放入r10
```

```

    addq %r10, %rax      # 将当前节点的值加到返回值中
    mrmovq 8(%rdi), %rdi # 取出下一个节点的地址放入rdi中
    jmp Loop             # 继续循环
Done:
    ret

#-----
# 链表元素示例
#-----

    .align 8
ele1:
    .quad 0x00a          # long的大小为8字节 (.quad), 0x00a是当前节点的值
    .quad ele2           # 下一个节点的地址
ele2:
    .quad 0x0b0
    .quad ele3
ele3:
    .quad 0xc00
    .quad 0

#-----
# 栈空间
#-----

    .pos 0x200
stack:

```

下面给出sum.yo

```

- |
  | #-----
  |
  | # sum.js
  | # Name:徐文博
  | # 学号:1320240207

```

```

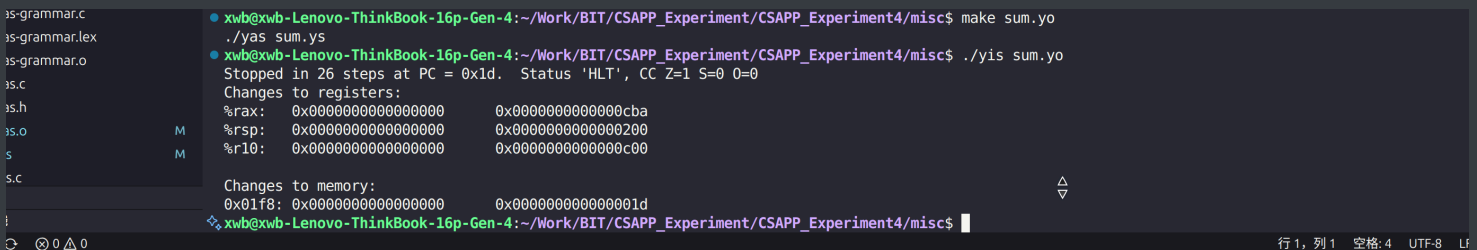
| # 链表元素求和
| #-----
-
|
0x000: | .pos 0
0x000: 30f4000200000000000000 | irmovq stack, %rsp # 初始化栈指针
0x00a: 30f7500000000000000000 | irmovq ele1, %rdi # 传入第一个参数, 链表头
第一个元素地址
0x014: 801e000000000000000000 | call sum_list # 调用sum_list函数
0x01d: 00 | halt # 程序结束
|
| #-----
-
| # long sum_list(list_ptr ls)
| # %rdi = ls
| # 返回指针在%rax中
| #-----
-
0x01e: | sum_list:
0x01e: 6300 | xorq %rax, %rax #作为返回结果
|
0x020: | Loop:
0x020: 6277 | andq %rdi, %rdi # 判断参数ls是否为NULL
0x022: 734a000000000000000000 | je Done # 如果是NULL, 跳转到Done
0x02b: 50a7000000000000000000 | mrmovq 0(%rdi), %r10 # 取出当前节点的8字节值并
放入r10
0x035: 60a0 | addq %r10, %rax # 将当前节点的值加到返回值
中
0x037: 5077080000000000000000 | mrmovq 8(%rdi), %rdi # 取出下一个节点的地址放入
rdi中
0x041: 7020000000000000000000 | jmp Loop # 继续循环
0x04a: | Done:
0x04a: 90 | ret
|
|
| #-----
-
| # 链表元素示例

```

```

- | #-----
0x050: | .align 8
0x050: | ele1:
0x050: 0a00000000000000 | .quad 0x00a # long的大小为8字节 (.quad),
0x00a是当前节点的值
0x058: 6000000000000000 | .quad ele2 # 下一个节点的地址
0x060: | ele2:
0x060: b000000000000000 | .quad 0x0b0
0x068: 7000000000000000 | .quad ele3
0x070: | ele3:
0x070: 000c000000000000 | .quad 0xc00
0x078: 0000000000000000 | .quad 0
|
|
| #-----
- |
| # 栈空间
| #-----
- |
0x200: | .pos 0x200
0x200: | stack:
|
```

下面附上运行截图



rsum.yo-对链表元素进行递归求和

这里比上一道需要注意的是，我们在递归时，第一个参数一定是存放在%rdi的，所以在递归之前 `long val = ls->val;`，只能用%r10存放节点数值，然后再压入栈中（因为这是调用者保存，后面我们递归调用rsum_list会覆盖寄存器%r10），而这里也需要两个ret，一个是在Empty链表时ret，一个是正常递归完ret避免顺序执行到Empty。

下面附上rsum.js代码

```
# rsum.js
# Name:徐文博
# 学号:1320240207

    .pos 0
    irmovq stack, %rsp    # 初始化栈指针
    irmovq ele1, %rdi     # 传入第一个参数, 链表头第一个元素地址
    call rsum_list        # 调用rsum_list函数
    halt                  # 程序结束

#-----
# long rsum_list(list_ptr ls)
# %rdi = ls
# 返回指保存在%rax中
#-----

rsum_list:
    # if (!ls) return 0;
    andq %rdi,%rdi        # 判断参数ls是否为NULL
    je Empty              # 如果是NULL, 跳转到Done, 即空链表

    # long val = ls->value;
    mrmovq 0(%rdi), %r10   # 取出当前节点的8字节值并放入r10
                           #这里不用%rdi,是因为下面还要用到%rdi取下一个节点地址
                           #并call时要用到%rdi传参
    pushq %r10             # 将当前节点的值压栈保存,调用者保存
                           # 以便下面递归调用后再取出使用

    # long rest = rsum_list(ls->next);
    mrmovq 8(%rdi),%rdi   # 取出下一个节点的地址放入rdi中
    call rsum_list        # 递归调用rsum_list

    # return val + rest;
    popq %r10
    addq %r10,%rax
    ret
```

```

Empty:
    xorq %rax,%rax      # 作为返回结果初始化为0
    ret

# 链表元素示例
    .align 8
ele1:
    .quad 0x00a         # long的大小为8字节 (.quad), 0x00a是当前节点的值
    .quad ele2          # 下一个节点的地址
ele2:
    .quad 0x0b0
    .quad ele3
ele3:
    .quad 0xc00
    .quad 0

# 栈区
    .pos 0x200
stack:

```

下面附上rsum.yo内容

```

                                | # rsum.yo
                                | # Name:徐文博
                                | # 学号:1320240207
                                |
                                |
0x000:                          | .pos 0
0x000: 30f4000200000000000000 | irmovq stack, %rsp      # 初始化栈指针
0x00a: 30f7500000000000000000 | irmovq ele1, %rdi      # 传入第一个参数, 链表头第
一个元素地址
0x014: 801e000000000000000000 | call rsum_list         # 调用rsum_list函数

```

```

0x01d: 00          |      halt                      # 程序结束
                    |
                    | #-----
-
                    | # long rsum_list(list_ptr ls)
                    | # %rdi = ls
                    | # 返回指保存在%rax中
                    | #-----
-
0x01e:           | rsum_list:
                    | # if (!ls) return 0;
0x01e: 6277       |      andq %rdi,%rdi           # 判断参数ls是否为NULL
0x020: 734d000000000000 |      je Empty                # 如果是NULL, 跳转到Done,
即空链表
                    |
                    | # long val = ls->value;
0x029: 50a7000000000000 |      mrmovq 0(%rdi), %r10     # 取出当前节点的8字节值并
放入r10
                    |
                    | #这里不用%rdi,是因为下面还要
用到%rdi取下一个节点地址
                    |
                    | #并call时要用到%rdi传参
0x033: a0af       |      pushq %r10              # 将当前节点的值压栈保存,调用
者保存
                    |
                    | # 以便下面递归调用后再取出使用
                    |
                    | # long rest = rsum_list(ls->next);
0x035: 5077080000000000 |      mrmovq 8(%rdi),%rdi     # 取出下一个节点的地址放入
rdi中
0x03f: 801e000000000000 |      call rsum_list          # 递归调用rsum_list
                    |
                    | # return val + rest;
0x048: b0af       |      popq %r10
0x04a: 60a0       |      addq %r10,%rax
0x04c: 90         |      ret
                    |
                    |
0x04d:           | Empty:
0x04d: 6300       |      xorq %rax,%rax          # 作为返回结果初始化为0

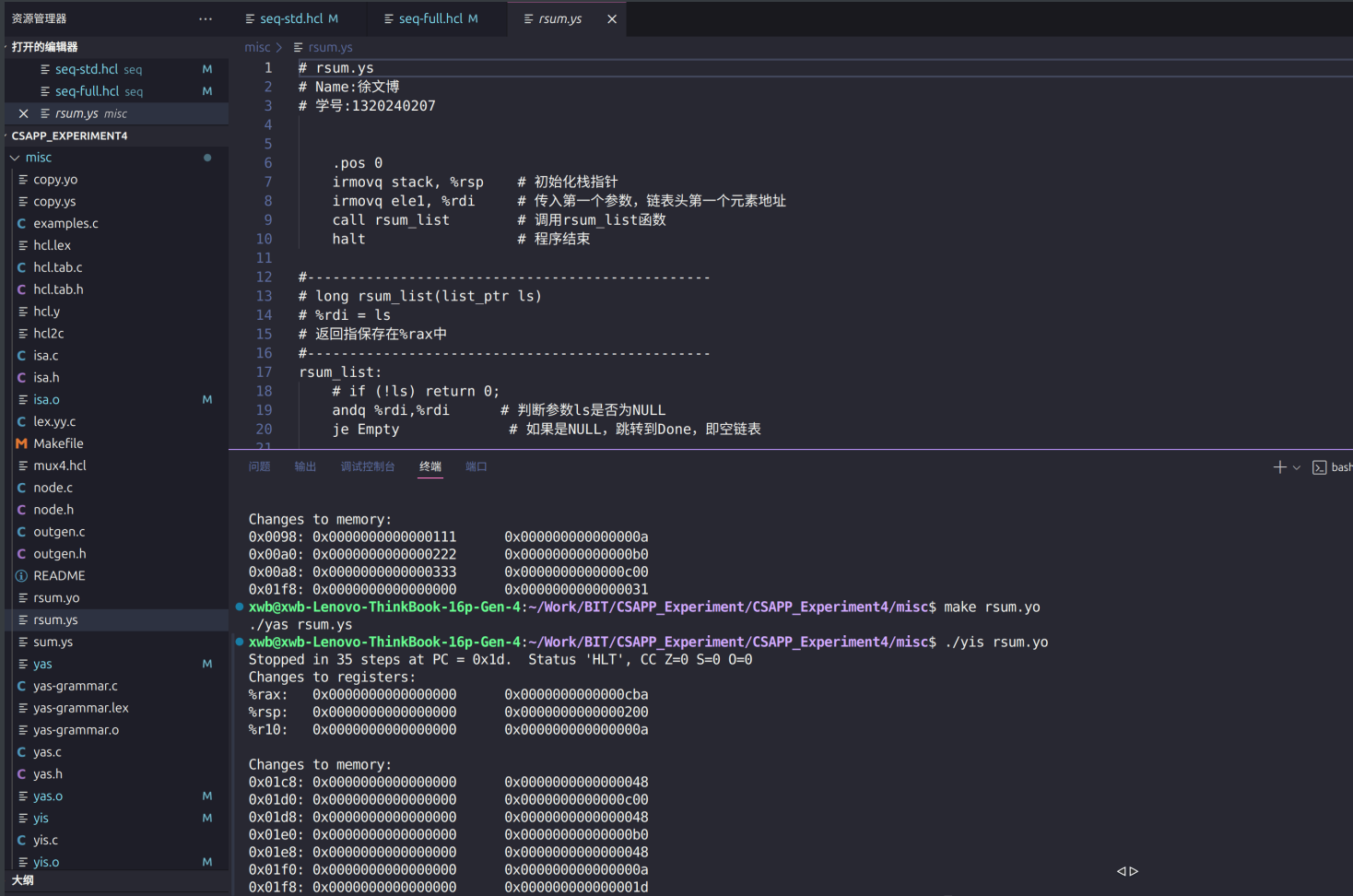
```

```

0x04f: 90          |      ret
                    |
                    |
                    | # 链表元素示例
0x050:          |      .align 8
0x050:          |  ele1:
0x050: 0a0000000000000000 |      .quad 0x00a      # long的大小为8字节 (.quad),
0x00a是当前节点的值
0x058: 600000000000000000 |      .quad ele2      # 下一个节点的地址
0x060:          |  ele2:
0x060: b00000000000000000 |      .quad 0x0b0
0x068: 700000000000000000 |      .quad ele3
0x070:          |  ele3:
0x070: 000c00000000000000 |      .quad 0xc00
0x078: 000000000000000000 |      .quad 0
                    |
                    |
                    | # 栈区
0x200:          |      .pos 0x200
0x200:          |  stack:
                    |

```

下面附上运行截图



copy.y-复制元素

下面附上代码以及运行效果

```
# copy.y
# Name: 徐文博
# 学号: 1320240207
# 链表元素复制
```

```
.pos 0
irmovq stack, %rsp      # 初始化栈指针
irmovq src, %rdi        # 传入第一个参数, Source链表头第一个元素地址
irmovq dest, %rsi       # 传入第二个参数, Dest链表头第一个元素地址
irmovq $3, %rdx         # 传入第三个参数, 链表元素个数
call copy_list          # 调用copy_list函数
halt                   # 程序结束
```

```

#-----
# void copy_list(list_ptr src, list_ptr dest, long n)
# %rdi = src
# %rsi = dest
# %rdx = n
#-----

copy_list:
    xorq %rax, %rax    # long result = 0;

Loop:
    andq %rdx, %rdx    # len <= 0?
    je Done            # if (len <= 0) break;

    mrmovq (%rdi), %r10 # long val = *src;
    rmmovq %r10, (%rsi) # *dest = val;
    xorq %r10, %rax     # result ^= val;

    irmovq $8, %r11 #因为long类型占8字节,
                    #因此src++和dest++都要加8
                    #Y86只支持寄存器相加
    addq %r11, %rdi     # src++;
    addq %r11, %rsi     # dest++;

    irmovq $1, %r11    # len--;
    subq %r11, %rdx

    jmp Loop

Done:
    ret

# Source链表和Dest链表的定义
    .align 8
src:
    .quad 0x00a
    .quad 0x0b0
    .quad 0xc00

```

```

dest:
    .quad 0x111
    .quad 0x222
    .quad 0x333

    .pos 0x200
stack:

```

下面是copy.yo

```

| # copy.js
| # Name:徐文博
| # 学号:1320240207
| # 链表元素复制
|
|
0x000: | .pos 0
0x000: 30f4000200000000000000 | irmovq stack, %rsp # 初始化栈指针
0x00a: 30f7800000000000000000 | irmovq src, %rdi # 传入第一个参数, Source链表
头第一个元素地址
0x014: 30f6980000000000000000 | irmovq dest, %rsi # 传入第二个参数, Dest链表头
第一个元素地址
0x01e: 30f2030000000000000000 | irmovq $3, %rdx # 传入第三个参数, 链表元素个
数
0x028: 8032000000000000000000 | call copy_list # 调用copy_list函数
0x031: 00 | halt # 程序结束
|
| #-----
-
| # void copy_list(list_ptr src, list_ptr dest,
long n)
| # %rdi = src
| # %rsi = dest
| # %rdx = n
| #-----
-

```

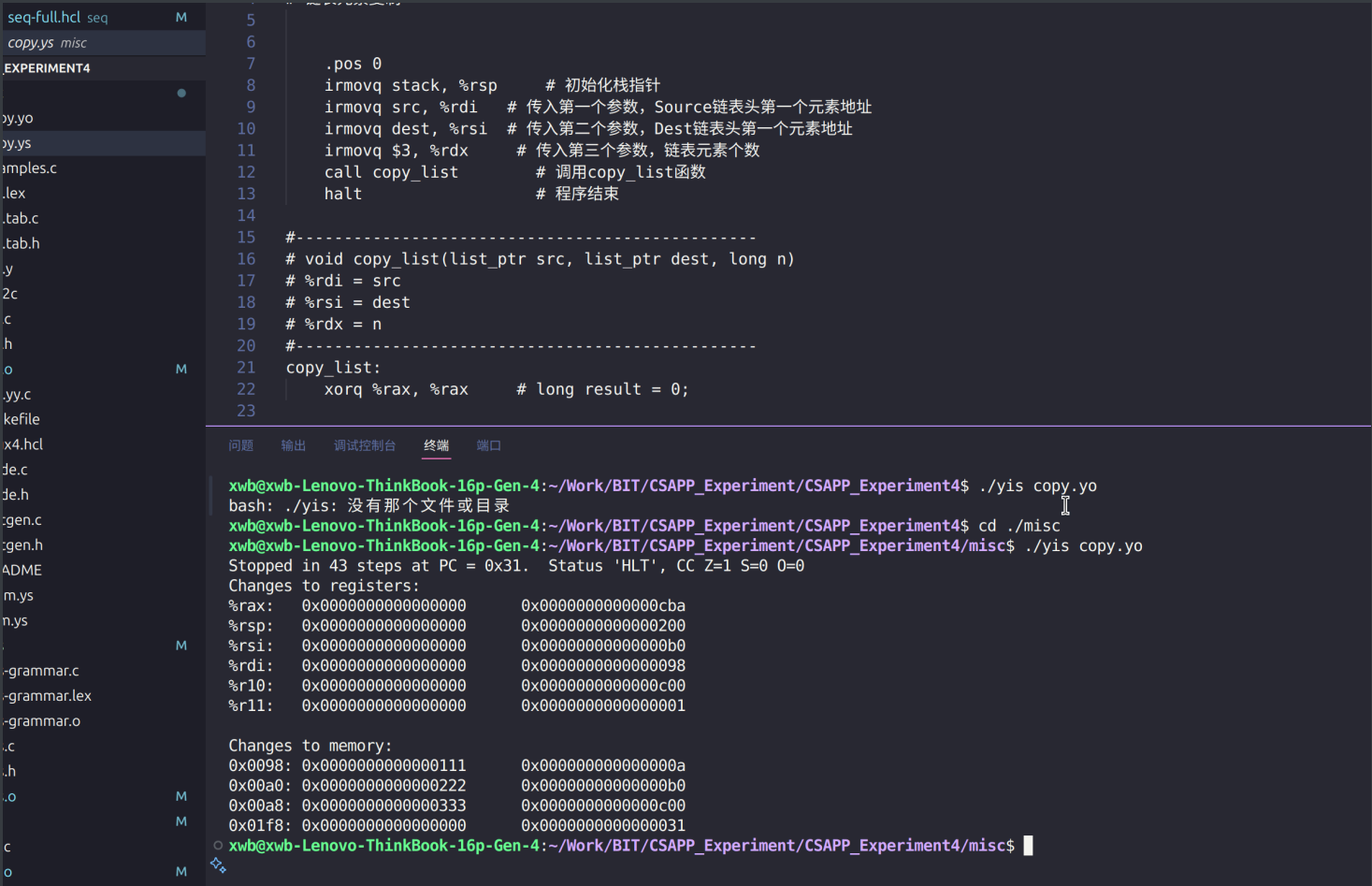
```

0x032:          | copy_list:
0x032: 6300      |     xorq %rax, %rax      # long result = 0;
                    |
0x034:          | Loop:
0x034: 6222      |     andq %rdx, %rdx      # len <= 0?
0x036: 737800000000000000 |     je Done              # if (len <= 0) break;
                    |
0x03f: 50a700000000000000 |     mrmovq (%rdi), %r10 # long val = *src;
0x049: 40a600000000000000 |     rmmovq %r10, (%rsi) # *dest = val;
0x053: 63a0      |     xorq %r10, %rax      # result ^= val;
                    |
0x055: 30fb08000000000000 |     irmovq $8, %r11 #因为long类型占8字节,
                    |                      #因此src++和dest++都要加8
                    |                      #Y86只支持寄存器相加
0x05f: 60b7      |     addq %r11, %rdi      # src++;
0x061: 60b6      |     addq %r11, %rsi      # dest++;
                    |
0x063: 30fb01000000000000 |     irmovq $1, %r11     # len--;
0x06d: 61b2      |     subq %r11, %rdx
                    |
0x06f: 703400000000000000 |     jmp Loop
                    |
0x078:          | Done:
0x078: 90        |     ret
                    |
                    | # Source链表和Dest链表的定义
0x080:          |     .align 8
0x080:          | src:
0x080: 0a0000000000000000 |     .quad 0x00a
0x088: b00000000000000000 |     .quad 0x0b0
0x090: 000c00000000000000 |     .quad 0xc00
0x098:          | dest:
0x098: 110100000000000000 |     .quad 0x111
0x0a0: 220200000000000000 |     .quad 0x222
0x0a8: 330300000000000000 |     .quad 0x333
                    |

```

```
0x200: | .pos 0x200
0x200: | stack:
```

下面则是截图



PartB

下面给出具体修改

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

bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
              IIRMOVQ, IRMMOVQ, IMRMVQ, IIADDQ };

bool need_valC =
```

```

        icode in { IIRMOVQ, IRMMOVQ, IMRM MOVQ, IJXX, ICALL, IIADDQ };

word srcB = [
    icode in { IOPQ, IRMMOVQ, IMRM MOVQ, 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, IMRM MOVQ, IIADDQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];

word aluB = [
    icode in { IRMMOVQ, IMRM MOVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
    icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Other instructions don't need ALU
];

```

下面附上ISA测试截图

5

CSAPP_Experiment4

拼 人 无线 音量 电池

seq-std.hcl M seq-full.hcl M X

seq > seq-full.hcl

```
38 wordsig IOPQ      'I_ALU'
39 wordsig IJXX      'I_JMP'
40 wordsig ICALL     'I_CALL'
41 wordsig IRET      'I_RET'
42 wordsig IPUSHQ    'I_PUSHQ'
43 wordsig IPOPQ     'I_POPQ'
44 # Instruction code for iaddq instruction
45 wordsig IIADDQ     'I_IADDQ'
46
47 ##### Symbolic representations of Y86-64 function codes #####
48 wordsig FNONE     'F_NONE'      # Default function code
49
50 ##### Symbolic representation of Y86-64 Registers referenced explicitly #####
51 wordsig RRSP      'REG_RSP'      # Stack Pointer
52 wordsig RNONE     'REG_NONE'     # Special value indicating "no register"
53
54 ##### ALU Functions referenced explicitly #####
```

问题 输出 调试控制台 终端 端口

+ v bash - seq 窗口 更多 全屏 退出

xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/seq\$./ssim -t ../y86-code/asumi.yo

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
0x00f8: 0x0000000000000000
ISA Check Succeeds

存在可用更新。

下载更新 稍后 发行说明

xwb@xwb-Lenovo-ThinkBook-16p-Ge

main* 0 0 Wooden-Flute (2 天前) 行 12, 列 21 (已选择6) 制表符长度: 4 UTF-8 LF {} 纯文本

回归测试原有ISA截图

```
50 ##### Symbolic representation of Y86-64 Registers referenced explicitly #####
51 wordsig RRSP      'REG_RSP'      # Stack Pointer
52 wordsig RNONE     'REG_NONE'     # Special value indicating "no register"
53
54 ##### All Functions referenced explicitly #####
```

问题 输出 调试控制台 终端 端口 + v bash - seq [] [] ... [] [] x

- **xwb@xwb-Lenovo-ThinkBook-16p-Gen-4**:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/seq\$
(cd ../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
- **xwb@xwb-Lenovo-ThinkBook-16p-Gen-4**:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/seq\$

回归测试iaddq截图

```
All 600 ISA Checks Succeed
```

- **xwb@xwb-Lenovo-ThinkBook-16p-Gen-4**:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/seq\$
(cd ../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
- **xwb@xwb-Lenovo-ThinkBook-16p-Gen-4**:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/seq\$

PartC

简述

讲真的，一下子看完archlab-cn.pdf的实验说明，没看懂，只知道有 `ncopy.js`，`pipe-full.hcl`，`sdriver.yo`和`ldriver.yo`，`correctness.pl`和`benchmark.pl`，下面理一下思路。注意：下面的命令一切都是在`sim/pipe/`目录下执行。

1. `ncopy.js` -汇编程序

- 作用：这是我要实现的**Y86-64汇编程序**，其核心功能是：
 - 将一个数组 `src` 复制到另一个数组 `dst`
 - 计算并返回 `src` 中大于零的元素个数，最终这个结果会存储在 `%rax` 寄存器中。
- 目的：我需要优化它的**执行周期（CPE）**，确保它在不同输入规模下的执行效率和正确性。

2. `pipe-full.hcl` -流水线处理器的控制逻辑

- 作用：这是**流水线CPU的硬件控制逻辑**，我需要在這裡定义处理器如何执行 `ncopy.js` 中的指令。它决定了：
 - **流水线各阶段**（取值、译码、执行、访存、写回）如何工作。
 - 如何优化流水线中的**数据转发、分支预测、冒险控制**等。
- 和 `ncopy.js` 的关系：
 - `ncopy.js` 运行时会依赖 `pipe-full.hcl` 里的控制逻辑，流水线的控制逻辑会影响 `ncopy.js` 的执行效率。

3. `sdriver.yo` 和 `ldriver.yo` -测试程序

- 作用：
 - `sdriver.yo`：测试 `ncopy.js` 在一个小数组（4个元素）上的正确性。如果 `ncopy.js` 正确执行，它会在 `%rax` 中返回2。
 - `ldriver.yo`：测试 `ncopy.js` 在一个大数组（63个元素）上的正确性。如果 `ncopy.js` 正确执行，那么它会在 `%rax` 中返回31。

4. `correctness.pl` -一个更严格的正确测试

- 作用：这是一个**自动化测试工具**，用来确保 `ncopy.js` 在各种不同大小的数组上的正确性。
- 测试内容：
 - 它会自动生成多种不同大小的数组，并测试 `ncopy.js` 函数是否正确复制数据并计算正整数个数。

5. benchmark.pl -性能测试工具

- 作用：顾名思义，用来测试 `ncopy.js` 的性能
- 测试内容：
 - 它会运行 `ncopy.js` 函数，计算并复制每一个元素需要多少周期（CPE）。
 - 目标是 $CPE < 9$ 。

下面我打算在修改之前，我先跑一遍，下面给出命令

1. `make drivers`：是用来生成测试驱动程序的，执行后会自动生成并汇编两个 `.yo` 文件（`sdriver.yo` 和 `ldriver.yo`），分别用于小数组和大数组的测试。
2. `make psim VERSION=full`：这个命令会重新编译**流水线模拟器（PSIM）**，编译时会由 `pipe-full.hcl` 生成对应的 `pipe-full.c`（HCL→C），再与 `psim.c`、`isa.c` 等一起编译生成可执行文件 `psim`。

注意： `make VERSION=full = make drivers + make psim VERSION=full`

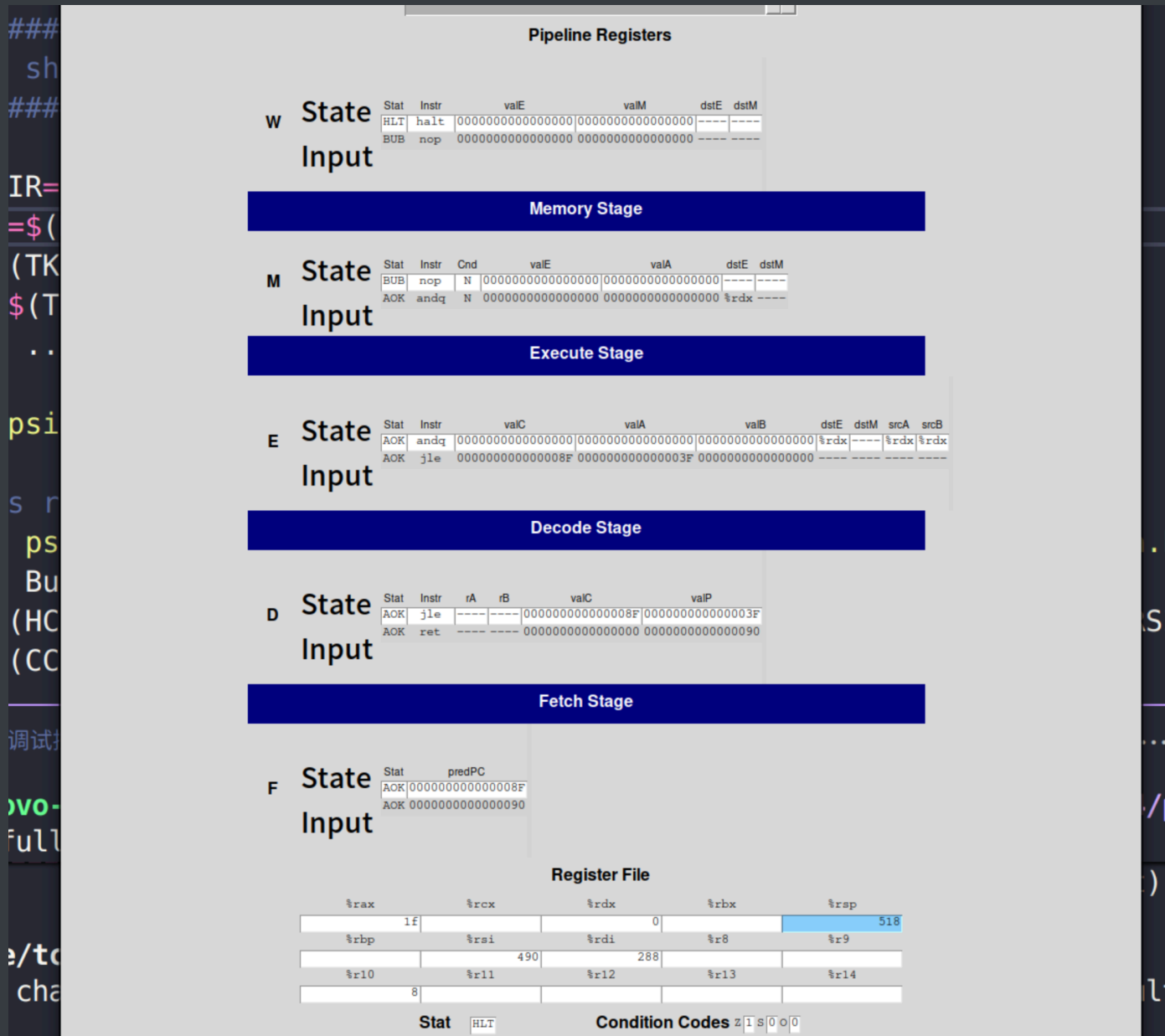
3. `./psim -g sdriver.yo`：`sdriver.yo` 是一个测试小数组（4个元素）的程序，它会调用我们编写的 `ncopy.js` 函数执行数组复制，并返回正整数的数量（即 `%rax` 中的值）。

`sdriver.yo` / `ldriver.yo` 是驱动程序；用 `psim` 运行时体现流水线执行，用 `ysis` 运行时体现顺序执行。

我这里第一次运行时，出了一个问题，可能是没有执行 `make clean` 清理之前的编译缓存。因此我执行了 `make clean`，然后执行 `make VERSION=full`，再运行 `./psim -g sdriver.yo` 则成功出现了GUI页面，也运行成功。

4. `./psim -g ldriver.yo`：是用来 **测试大数组（63 个元素）** 的，它会启动 **PSIM 模拟器**（流水线）并加载之前生成的 `ldriver.yo` 驱动程序文件，进入 **GUI 模式** 来运行模拟。

下面附上截图



5. `../misc/yis sdriver.yo`：该指令会启动YIS模拟器（顺序执行），并加载 `sdriver.yo` 驱动程序。同样通过检查 `%rax` 的最终值是否正确。这回和之前的 `../psim` 不同，这个是使用YIS模拟器，`../psim -g sdriver.yo` 是利用我们自己的pipe模拟器，YIS模拟器是来判断 `ncopy.yo` 汇编的正确性。

下面附上截图

```

xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$ ..
/misc/yis sdriver.yo
Stopped in 58 steps at PC = 0x31. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000      0x0000000000000002
%rsp: 0x0000000000000000      0x0000000000000170
%rsi: 0x0000000000000000      0x00000000000000e8
%rdi: 0x0000000000000000      0x00000000000000b0
%r10: 0x0000000000000000      0x0000000000000008

Changes to memory:
0x00c8: 0x0000000000cdefab      0xffffffffffffffff
0x00d0: 0x0000000000cdefab      0x0000000000000002
0x00d8: 0x0000000000cdefab      0xffffffffffffffff
0x00e0: 0x0000000000cdefab      0x0000000000000004
0x0168: 0x0000000000000000      0x0000000000000031
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$

```

6. correctness.pl : 是一个脚本, 它会生成一系列的测试程序, 每个程序的块从0到65, 再加上一些更大的块长度 (块长度就是我们需要复制的数组的长度, 例如4个元素, 63ge元素)。这个脚本提供了比 sdriver.yo 和 ldriver.yo 更严格的测试, 同时**结果会随机变化** (每次测试, 数组中的个数可能不同), 这会验证程序的准确度和稳定性。

具体操作步骤

- 运行 correctness.pl : 它会生成多个测试程序, 自动执行并验证 ncopy.yo 是否正确。
- 如果在运行 correctness.pl 后发现某个特定长度的测试失败, 可以用 gen-driver.pl 脚本来生成特定长度的**自定义驱动程序**。
- 假设某个长度为 **K** 的测试失败, 你可以通过以下命令生成自定义的驱动程序:

```

./gen-driver.pl -f ncopy.yo -n K -rc > driver.yo //生成自定义的驱动程序
make driver.yo //将生成的driver.yo编译为.yo文件
../misc/yis driver.yo //使用YIS模拟器来运行这个编译后的驱动程序

```

- 检查 %rax 寄存器的值:

在测试过程中, YIS 模拟器会根据测试结果将值存储在 %rax 寄存器中。我们需要检查 %rax 的值来判断测试是否成功:

0xaaaa : 所有测试通过, 程序没有错误。

0xbbbb : 计数不正确, 说明 ncopy.yo 函数返回的正整数个数有误。

0xcccc : ncopy 函数的长度超过了 1000 字节, 可能是因为你的程序写得过长, 需要优化。

0xdddd : 某些源数据没有被正确复制到目标数组。

0xeeee : 目标区域前后有数据被损坏 (超出了目标数组的边界)。

下面附上截图

```
pipe > M Makefile
13
问题 输出 调试控制台 终端 端口
31 OK
32 OK
33 OK
34 OK
35 OK
36 OK
37 OK
38 OK
39 OK
40 OK
41 OK
42 OK
43 OK
44 OK
45 OK
46 OK
47 OK
48 OK
49 OK
50 OK
51 OK
52 OK
53 OK
54 OK
55 OK
56 OK
57 OK
58 OK
59 OK
60 OK
61 OK
62 OK
63 OK
64 OK
128 OK
192 OK
256 OK
68/68 pass correctness test
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$
```

7. (cd ../y86-code; make testpsim)：这是基准测试，是利用sim/y86-code里面的Y86-64寄存器程序进行测试，其实就是测试：我们改了 pipe-full.hcl 以后，有没有把CPU的其他行为搞坏。这个和第8点的回归测试目的一样（只不过回归测试更全面）。下面附上截图

```
2 #####
问题 输出 调试控制台 终端 端口
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$ (cd ../y86-code; make testpsim)
Makefile:42: 警告: 忽略后继规则定义的先决条件
Makefile:45: 警告: 忽略后继规则定义的先决条件
Makefile:48: 警告: 忽略后继规则定义的先决条件
Makefile:51: 警告: 忽略后继规则定义的先决条件
../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 pushquestion.pipe pushtest.pipe prog1.pipe prog2.pipe prog3.pipe prog4.pipe prog5.pipe prog6.pipe prog7.pipe prog8.pipe ret-hazard.pi
pe
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$
```

8. (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)：这是进行回归测试：也是测试，我们的 pipe-full.hcl 逻辑是否正确。下面附上截图

```
2 #####
问题 输出 调试控制台 终端 端口 + v bash-pipe
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$ (c
d ../ptest; make SIM=../pipe/psim TFLAGS=-i)
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
Test op-iaddq-256-rdx failed ...
Test ji-jmp-32-64 failed
Test ji-jmp-64-32 failed
Test ji-jmp-64-64 failed
Test ji-jle-32-32 failed
Test ji-jle-32-64 failed
Test ji-jle-64-32 failed
Test ji-jle-64-64 failed
Test ji-jl-32-32 failed
Test ji-jl-32-64 failed
Test ji-jl-64-32 failed
Test ji-jl-64-64 failed
Test ji-je-32-32 failed
Test ji-je-32-64 failed
Test ji-je-64-32 failed
Test ji-je-64-64 failed
Test ji-jne-32-32 failed
Test ji-jne-32-64 failed
Test ji-jne-64-32 failed
Test ji-jne-64-64 failed
Test ji-jge-32-32 failed
Test ji-jge-32-64 failed
Test ji-jge-64-32 failed
Test ji-jge-64-64 failed
Test ji-jg-32-32 failed
Test ji-jg-32-64 failed
Test ji-jg-64-32 failed
Test ji-jg-64-64 failed
Test ji-call-32-32 failed
Test ji-call-32-64 failed
Test ji-call-64-32 failed
Test ji-call-64-64 failed
32/96 ISA Checks Failed
./ctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 22 ISA Checks Succeed
./hctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 756 ISA Checks Succeed
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pipe$
```

这里有错误很正常，因为 `TFLAGS=-i` 会另外测试我们在 `pipe-full.hcl` 里面添加的 `iaddq`，但现在我只是在跑通，所以很正常，如果我们执行 `(cd ../ptest; make SIM=../pipe/psim)`，那么将不会有问題。

9. `./correctness.pl -p`：这条指令和 `./correctness` 很像，但不同在于，`./correctness.pl` 测的是 `ncopy.py` 在顺序执行下是否正确，`correctness.pl -p` 测的是 `ncopy` 在「流水线执行」下是否正确。下面附上截图

```
323 # Conditions for a load/use hazard
问题 输出 调试控制台 终端 端口
+ v bash-pipe
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pip
e$ ./correctness.pl -p
29 OK
30 OK
31 OK
32 OK
33 OK
34 OK
35 OK
36 OK
37 OK
38 OK
39 OK
40 OK
41 OK
42 OK
43 OK
44 OK
45 OK
46 OK
47 OK
48 OK
49 OK
50 OK
51 OK
52 OK
53 OK
54 OK
55 OK
56 OK
57 OK
58 OK
59 OK
60 OK
61 OK
62 OK
63 OK
64 OK
128 OK
192 OK
256 OK
68/68 pass correctness test
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pip
e$
```

10. benchmark.pl : benchmark.pl 会用流水线 CPU (psim) 在大规模 ncopy 上测：平均每复制一个元素要多少个时钟周期 (CPE)。下面附上截图

```
323 # Conditions for a load/use hazard
问题 输出 调试控制台 终端 端口
+ v bash-pipe
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pip
e$ ./benchmark.pl
27      393      14.56
28      409      14.61
29      421      14.52
30      437      14.57
31      449      14.48
32      465      14.53
33      477      14.45
34      493      14.50
35      505      14.43
36      521      14.47
37      533      14.41
38      549      14.45
39      561      14.38
40      577      14.43
41      589      14.37
42      605      14.40
43      617      14.35
44      633      14.39
45      645      14.33
46      661      14.37
47      673      14.32
48      689      14.35
49      701      14.31
50      717      14.34
51      729      14.29
52      745      14.33
53      757      14.28
54      773      14.31
55      785      14.27
56      801      14.30
57      813      14.26
58      829      14.29
59      841      14.25
60      857      14.28
61      869      14.25
62      885      14.27
63      897      14.24
64      913      14.27
Average CPE      15.18
Score      0.0/60.0
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/CSAPP_Experiment/CSAPP_Experiment4/pip
e$
```

正常，我还没修改自然是0分。

简述小结

因此我们知道了，目前看来，ncopy.js 是正确的，只是效率不高

1. 目的：

- 修改 ncopy.js ，目的是让其效率更高

- 修改 pipe-full.hcl，目的是加入 IIADDQ 同时效率更高。

2. 各个测试对比：

- sdriver.yo 和 ldriver.yo 是用来测试 ncopy.yo 的正确性
- correctness.pl 是使用 YIS（顺序ISA模拟器）在多种不同规模下测试 ncopy.yo，进行正确性测试。
- correctness.pl -p：与 correctness.pl 不同在于，流水线执行正确性测试
- testpsim：基准程序测试，使用 sim/y86-code 中提供的标准 Y86-64 程序；验证修改 pipe-full.hcl 后，处理器是否仍能正确执行“非 ncopy”程序；用于检测是否引入了影响全局行为的错误。
- ptest：回归测试，对流水线处理器进行更全面、系统性的指令级回归测试；覆盖算术指令、跳转指令、调用与返回等多种场景；若未实现 iaddq，需避免使用 TFLAGS=-i 选项。
- benchmark.pl：性能测试，在确认所有正确性测试通过后运行；使用流水线处理器 psim 测量 ncopy.yo 的执行性能；以 CPE（Cycles Per Element）作为评价指标，反映程序在流水线下的平均执行效率。

注意：本实验中，同一个 ncopy.yo 程序，会在两种不同的处理器模型上被反复执行和验证：

- ../misc/yis：顺序处理器（YIS）
- ../psim：流水线执行（PSIM）测试

阅读 ncopy.yo 和 pipe-full.hcl

阅读ncopy.yo

```
#!/* $begin ncopy-yo */
#####
# ncopy.yo - 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.
# Name:徐文博
# ID:1320240207
# 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 */

```

▪ 入口约定：

- 参数： %rdi = src ， %rsi = dst ， %rdx = len
- 返回值放 %rax

- Loop header: 初始化+处理len<=0
- Loop主体: 每次处理1个元素
 - 从src读元素, 然后复制到dst中
 - 判断正数, 是正数, count(%rax)++
 - 更新len, src, dst。
- Loop循环条件: len > 0
- Done: 返回。

阅读pipe-full.hcl

前面, 那一大段都是在讲引用的库, 以及声明好的指令 (例如, INOP, IHALT等等), 还有一些默认的指令和状态 (冒泡呀之类的)。

需要注意以下几点

1. wordsig F_predPC 'pc_curr->pc' # Predicted value of PC : F_predPC =预测的下一条指令PC
2. ##### Intermediate Values in Fetch Stage #####

 wordsig imem_icode 'imem_icode' # icode field from instruction memory
 wordsig imem_ifun 'imem_ifun' # ifun field from instruction memory
 wordsig f_icode 'if_id_next->icode' # (Possibly modified) instruction code
 wordsig f_ifun 'if_id_next->ifun' # Fetched instruction function
 wordsig f_valC 'if_id_next->valc' # Constant data of fetched instruction
 wordsig f_valP 'if_id_next->valp' # Address of following instruction
 boolsig imem_error 'imem_error' # Error signal from instruction memory
 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?

这块定义的是在取值阶段需要使用到的**组合逻辑信号**

- imem_icode / imem_ifun 是**刚刚从指令存储器读出来的原始指令**
- f_icode / f_ifun 是准备写进Decode阶段寄存器的指令, 是这一拍 (拍, 指的是一个时间

点) 取值阶段得到的指令。

- `f_valC` : 这一拍 Fetch 阶段算出来的立即数, 准备交给 Decode
- `f_valP` : 顺序执行时的下一条 PC (PC+指令长度), 也是 **供 Decode / Execute 使用 (如果是call指令, Execute的valA就是此时的valP, 用来得到call跳转的地址)**。
- `imem_error` : 取值是否发生地址错误, **Fetch 阶段不会立刻停机**, 而是把错误转化为 `f_stat = SADR` , 一路送进流水线。
- `instr_valid` : 这条指令的 icode 是否合法, 会改变 `f_state`。

```
3. ##### Pipeline Register D #####  
wordsig D_icode 'if_id_curr->icode'    # Instruction code  
wordsig D_rA 'if_id_curr->ra'           # rA field from instruction  
wordsig D_rB 'if_id_curr->rb'           # rB field from instruction  
wordsig D_valP 'if_id_curr->valp'       # Incremented PC
```

这些就像之前一样顾名思义, 不过 `D_icode` 和 `d_icode` 是有区别的, `D` 代表的是在 Decode 这一硬件里面存在的 (用来保存这一拍 Decode 的状态), `d` 代表的是要传给下个阶段的值 (是一个组合逻辑信号, Decode 就是传给 Execute)。

- `D_icode` : Decode 阶段正在处理的那条指令的 **指令类型**, 来源于上一拍的 `f_icode` 。
- `D_rA` : 指令编码里的 `rA` 的字段。
- `D_rB` : 同 `D_rA` 。
- `D_valP` : 来源上一拍的 `f_icode` , 用途如下
 - `call` : 把 `D_valP` 当返回地址压栈。
 - `jxx` : 条件不成立时回到 `D_valP` 。
 - `ret` : 用于 PC 修正。

注意: 这里 `f_ifun` 会被 IF/ID 寄存器送到 Eexecute (便于 E 阶段处理 OPQ, JXX, RRMVQ (条件))。

```
4. ##### Intermediate Values in Decode Stage #####  
  
wordsig d_srcA 'id_ex_next->srca'    # srcA from decoded instruction  
wordsig d_srcB 'id_ex_next->srcb'    # srcB from decoded instruction  
wordsig d_rvalA 'd_regvala'         # valA read from register file  
wordsig d_rvalB 'd_regvalb'         # valB read from register file
```

这里就不多做解释, 就是注意 `D` 和 `d` 的区别, `db` 表明这些物理上都不是存在的, 是组合逻辑信号 (Decode 得到然后传送到 Execute), 用来传递到 Execute, 比如 `d_rvalA`, 就是 Decode 得到的,

然后传到Execute，用来的到E_valA（假如它需要）。我们后面hcl要写的控制逻辑就是改写这些东西。

```
5. ##### Pipeline Register E #####
wordsig E_icode 'id_ex_curr->icode'    # Instruction code
wordsig E_ifun  'id_ex_curr->ifun'      # Instruction function
wordsig E_valC  'id_ex_curr->valc'      # Constant data
wordsig E_srcA  'id_ex_curr->srca'      # Source A register ID
wordsig E_valA  'id_ex_curr->vala'      # Source A value
wordsig E_srcB  'id_ex_curr->srcb'      # Source B register ID
wordsig E_valB  'id_ex_curr->valb'      # Source B value
wordsig E_dstE  'id_ex_curr->deste'     # Destination E register ID
wordsig E_dstM  'id_ex_curr->destm'     # Destination M register ID
```

这里一句话概括了：Pipeline Register E（ID/EX）保存的是：Decode 阶段已经“决定好的一切”，供 Execute 阶段在下一拍直接使用。e_valE等等e开头的，就是通过这些E_valA和E_valB等等得到的。其他的Decode,Memory同理。

```
6. ##### Intermediate Values in Execute Stage #####
wordsig e_valE 'ex_mem_next->vale'     # vale generated by ALU
boolsig e_Cnd  'ex_mem_next->takebranch' # Does condition hold?
wordsig e_dstE 'ex_mem_next->deste'     # dstE (possibly modified to be RNONE)
```

Execute 阶段“组合逻辑算出来的结果”，它们将在本拍末尾被写入 EX/MEM 流水线寄存器，供 Memory 阶段以及后续阶段使用。

```
7. ##### Pipeline Register M #####
wordsig M_stat 'ex_mem_curr->status'    # Instruction status
wordsig M_icode 'ex_mem_curr->icode'     # Instruction code
wordsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
wordsig M_valA  'ex_mem_curr->vala'      # Source A value
wordsig M_dstE  'ex_mem_curr->deste'     # Destination E register ID
wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
wordsig M_dstM  'ex_mem_curr->destm'     # Destination M register ID
boolsig M_Cnd   'ex_mem_curr->takebranch' # Condition flag
boolsig dmem_error 'dmem_error'         # Error signal from instruction memory
```

```
##### Intermediate Values in Memory Stage #####
wordsig m_valM 'mem_wb_next->valm' # valM generated by memory
wordsig m_stat 'mem_wb_next->status' # stat (possibly modified to be
SADR)

##### Pipeline Register W #####
wordsig W_stat 'mem_wb_curr->status' # Instruction status
wordsig W_icode 'mem_wb_curr->icode' # Instruction code
wordsig W_dstE 'mem_wb_curr->deste' # Destination E register ID
wordsig W_valE 'mem_wb_curr->vale' # ALU E value
wordsig W_dstM 'mem_wb_curr->destm' # Destination M register ID
wordsig W_valM 'mem_wb_curr->valm' # Memory M value
```

这里只说明，比较特别的

- dmem_error —— 内存访问是否出错

下面就是控制逻辑

Fetch Stage

```
##### Fetch Stage #####

## What address should instruction be fetched at
word f_pc = [
    # Mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    # Completion of RET instruction
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];

## Determine icode of fetched instruction
word f_icode = [
    imem_error : INOP;
    1: imem_icode;
];
```

```

# Determine ifun
word f_ifun = [
    imem_error : FNONE;
    1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMVQ,
      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

# Determine status code for fetched instruction
word f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];

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

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

# Predict next value of PC
word f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

```

1. 选择PC来源

- `M_icode == IJXX && !M_Cnd`: `M_valA`: 之前取过的分支指令，现在发现“没跳”；那之前猜错了；用顺序PC（存在`M_valA`里）纠正。

- `W_icode == IRET` : `W_valM` : `ret` 的返回地址，只有到W阶段才知道；一旦知道，立刻用它。
 - 1: `F_predPC` : 没事，继续用之前“猜的PC”。
2. 从内存中读取指令以及状态: `f_icode` , `f_ifun` , `f_state` , `instr_valid` 。
 3. 根据指令，判断是否需要寄存器，立即数: `need_regids` , `need_valC` 。
 4. 预测下一拍的指令: `f_predPC` 。

Decode Stage

```
## What register should be used as the A source?
word d_srcA = [
    D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : D_rA;
    D_icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word d_srcB = [
    D_icode in { IOPQ, IRMMOVQ, IMRMVQ } : D_rB;
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't need register
];

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

## What register should be used as the M destination?
word d_dstM = [
    D_icode in { IMRMVQ, IPOPQ } : D_rA;
    1 : RNONE; # Don't write any register
];

## What should be the A value?
```

```

## Forward into decode stage for valA
word d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    d_srcA == e_dstE : e_valE;      # Forward valE from execute
    d_srcA == M_dstM : m_valM;      # Forward valM from memory
    d_srcA == M_dstE : M_valE;      # Forward valE from memory
    d_srcA == W_dstM : W_valM;      # Forward valM from write back
    d_srcA == W_dstE : W_valE;      # Forward valE from write back
    1 : d_rvalA; # Use value read from register file
];

word d_valB = [
    d_srcB == e_dstE : e_valE;      # Forward valE from execute
    d_srcB == M_dstM : m_valM;      # Forward valM from memory
    d_srcB == M_dstE : M_valE;      # Forward valE from memory
    d_srcB == W_dstM : W_valM;      # Forward valM from write back
    d_srcB == W_dstE : W_valE;      # Forward valE from write back
    1 : d_rvalB; # Use value read from register file
];

```

首先这里的 `D_...` 都来自 **IF/ID (D阶段流水线寄存器)**，也就是“当前 Decode 正在处理的那条指令”的字段：

- `D_icode`：指令类型
- `D_rA` / `D_rB`：指令编码里写的寄存器号（ra/rb 字段）
- `D_valP`：这条指令的顺序下一条 PC（PC+指令长度）

Decode 阶段会产出（写入 ID/EX，也就是 E 寄存器）：

- `d_srcA` / `d_srcB`：这条指令**需要读取**的源寄存器编号
- `d_dstE` / `d_dstM`：这条指令**将来要写回**的目的寄存器编号（E□ / M□）
- `d_valA` / `d_valB`：真正给 Execute 用的两个操作数值（**已经做过转发**）

还有两个是“从寄存器堆读出来的原始值”：

- `d_rvalA`：如果按 `d_srcA` 去寄存器文件读，读出来的值

- d_rvalB：如果按 d_srcB 去寄存器文件读，读出来的值
1. d_srcA：判断需要从哪个寄存器读取"A操作数"
 2. d_srcB：判断需要从哪个寄存器读取"B操作数"
 3. d_dstE：判断将来把ALU结果写到哪个寄存器
 4. d_dstM：判断将来把“内存读出来的值”写到哪个寄存器
 5. d_valA：A的值怎么拿
 - ICALL,IJXX：此时valA不是寄存器的值，而是valP（即顺序执行的下一条指令地址），call 需要把返回地址压入栈，返回地址就是 D_valP； jXX 有时也需要valP（假设jXX条件判断错误），所以val=D_valP。
 - 数据转发（顺着优先级，从“最新”到“最旧”）

如果 d_srcA 正好等于后面某条指令即将写的寄存器（触发数据转发）：

 - == e_dstE：说明执行阶段刚算出来（最新）->用 e_valE。
 - == M_dstM：内存阶段刚读出来->用 m_valM
 - == M_dstE：内存阶段有ALU结果->用 M_valE
 - == W_dstE，== W_dstM：写回阶段最终值，分别用-> W_valE，W_valM。
 - 1:d_rvalA：以上都不命中，那么就从寄存器中读取。
 6. d_valB：同理，只是没有 call / jXX 的特例，因为 call / jXX 根本不使用B操作数。

Execute Stage

```
##### Execute Stage #####
```

```
## Select input A to ALU
```

```
word aluA = [
```

```
    E_icode in { IRRMOVQ, IOPQ } : E_valA;
```

```
    E_icode in { IIRMOVQ, IRMMOVQ, IMRMVQ } : E_valC;
```

```
    E_icode in { ICALL, IPUSHQ } : -8;
```

```
    E_icode in { IRET, IPOPOP } : 8;
```

```
    # Other instructions don't need ALU
```

```
];
```

```
## Select input B to ALU
```

```
word aluB = [
```

```

    E_icode in { IRMMOVQ, IMRMovQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ } : E_valB;
    E_icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
    E_icode == IOPQ : E_ifun;
    1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

## Generate valA in execute stage
word e_valA = E_valA;    # Pass valA through stage

## Set dstE to RNONE in event of not-taken conditional move
word e_dstE = [
    E_icode == IRRMOVQ && !e_Cnd : RNONE;
    1 : E_dstE;
];

```

1. aluA：只是根据指令选择，哪一根线接到ALU的A端
 - IRRMOVQ / IOPQ：aluA使用寄存器里的值，E_valA（Decode已准备好的A值）。
 - IIRMOVQ / IRMMOVQ / IMRMovQ：这些指令都需要立即数valC，
 - ICALL / PUSHQ：都需要 $rsp = rsp - 8$ ，A端固定 -8
 - IRET / IPOPQ：都要 $rsp = rsp + 8$ ，A端固定 8
2. aluB：B端通常是“基址/被累加的值”。
3. alufun：ALU做什么运算，只有 IOPQ（add/sub/and/xor）真正的运算是由 ifun 决定的，其余都是加法。
4. set_cc：这里很严格，只有 IOPQ 会更新条件码，出异常时不允许再更新条件码。
5. e_valA：这里将 E_valA 原样传下去，有下面两个用途

- `rrmovq`：Memory阶段要把 `valA` 写入内存，这里`valA`是寄存器`rA`的值
- `ret` / `pop`：Memory阶段要用 `valA` 作为访存的地址，从内存中取出传给PC/`rA`的值，这里`valA`是寄存器`rsp`的值。同时传给PC的话，就要修改之前预测的PC，返回去修改Fetch的 `f_PC`。
- `call`：Memory阶段要将`valA`压入栈（既内存中），这里的`valA`是Fetch阶段中的`valP`。
- `JXX`：传下去，是为了后续Execute得出 `e_Cnd` 之后，如果决定不跳，那么需要修改预测的PC，返回去修改Fetch的 `f_PC`。

6. `e_dstE`： `IRMMOVQ` 包括了 `rrmovq`（无条件），`comvle`，`comvg`（有条件），因此需要 `e_Cnd` 判断条件是否成立。其他时候都要传 `E_dstE`，便于后续数据转发。

Memory Stage

```
##### Memory Stage #####

## Select memory address
word mem_addr = [
    M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMVQ } : M_valE;
    M_icode in { IPOPQ, IRET } : M_valA;
    # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

/* $begin pipe-m_stat-hcl */
## Update the status
word m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];
```

1. `mem_addr`：访内存用哪个地址

- 使用 `M_valE` 的情况：这些指令的**内存地址**，都是在 **Execute** 阶段算出来的

- 使用 M_valA 的情况：popq rA，ret这两条指令的访问地址是addr=rsp，而rsp的旧值是在 **Execute阶段通过 E_valA 原样传下来的。**

2. mem_read：判断是否读内存

- IMRMOVQ：从内存读数据
- IPOPOPQ：从栈读数据
- IRET：从栈读返回地址。

3. mem_write：判断是否写内存

- IRMMOVQ：写数据到内存
- IPUSHQ：写栈
- ICALL：写返回地址

4. m_state：内存访问是否出错

Write Stage

```

/* $end pipe-m_stat-hcl */

## Set E port register ID
word w_dstE = W_dstE;

## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

## Update processor status
word Stat = [
    W_stat == SBUB : SAOK;
    1 : W_stat;
];

```

这里就只是将W寄存器里的东西接到"寄存器文件写口"

写回阶段有两种写寄存器来源：

- valE：ALU计算的结果
- valM：从内存读的结果

State则是整个CPU的状态。

Pipeline Register Control

1. 先理解Stall（停住），Bubble（塞气泡）

- Stall（停住）
 - 这个流水线寄存器 **本拍不更新**
 - 意味着：这一阶段的指令/数据 **继续保持上一拍的内容**
 - 用来“等一等”，别让错误继续传播
- Bubble（塞气泡）
 - 这个流水线寄存器 **本拍强行写入一个 NOP（STAT_BUB）**
 - 意味着：把某条“错误/不该执行的指令”冲掉
 - 用来“清洗流水线”(flush)

2. F寄存器控制

```
bool F_bubble = 0;
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRM0VQ, IPOPOQ } &&
    E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };
```

- F_bubble=0：PC没办法变为NOP，所以F只会stall
- F_stall 有两类情况
 - load/use冒险

条件：1.E阶段正在i执行 mrmovq 或 popq （从内存读数据**写到寄存器**）;2.并且E阶段要写的寄存器 E_dstM，正好是Decode阶段下一条指令要用的源寄存器 d_srcA/d_srcB。

例子

```
mrmovq 0(%rdi), %r10    # 这条在E阶段，结果要到M阶段才出来
addq    %r10, %rax       # 这条在D阶段，立刻要用 %r10
```

这时候必须要**Fetch停住**。

F stall 不是因为 Fetch 用不到值，而是因为：“**不能让错误的指令继续进入流水线**”。

- ret冒险

IRET in {D_icode, E_icode, M_icode}，因为ret的下一条指令要从栈里取出来，只有ret走到M/W（M结束，W开始得到valM）才知道PC。所以ret在流水线，必须停住**取值**，避免错误的指令进入流水线。

3. D寄存器控制

```
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } &&
    E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
    IRET in { D_icode, E_icode, M_icode };
```

- D_stall：只有在load/use时stall，原因和F一样，此时D阶段那条“用到load结果的指令”不能往前走。所以：**F stall + D stall** 一起发生。
- D_bubble：两种情况会bubble（冲掉Decode里的错误指令）

- A.分支预测失败—— E_icode == IJXX && !e_Cnd

E 阶段的 jxx 现在算出来条件不成立（ !e_Cnd ），说明之前 Fetch 按“跳转”预测取的指令是错的。

- B.ret冒险（不是load/use）—— !(load/use) && IRET in { D_icode, E_icode, M_icode }

ret期间我们要停住Fetch，而Decode里面那条指令此时往往是“**取错的/不该继续的**”，

所以策略是F stall（不再取新），D则bubble（把Decode清空，等ret把正确PC给出来）。

4. E寄存器控制

```
bool E_stall = 0;
bool E_bubble =
    (E_icode == IJXX && !e_Cnd) ||
    E_icode in { IMRM0VQ, IPOPOQ } &&
    E_dstM in { d_srcA, d_srcB };
```

- E_stall永远是0，E阶段不会选择停住，而是用 bubble 来解决问题。
- E_bubble两种情况
 - A.分支预测失败：当发现跳转失败，要清理掉D、E两级中的错误指令
 - B.load/use冒险：这是经典的策略：F stall + D stall + E bubble。这一拍结束后，E会被Bubble而计算出来的e_valE则会送到Memory，然后m_valM会被数据转发到Decode的d_valA/d_valB。

5. M寄存器

```
bool M_stall = 0;
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
```

- M_stall永远0：异常处理用bubble来做
- M_bubble：只要Memory阶段或Writeback阶段已经出现异常（地址错/非法指令/halt）就开始bubbleM。

6. W寄存器

```
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
```

- W_bubble永远0，W是最后一个阶段，不需要bubble
- W_stall：异常时停住。

修改pipe-full.hcl

很好，我们阅读完了，也大致有了思路，我先往pipe-full.hcl里面加入 IADDQ 指令，然后修改 ncpu.py 。

加入 IADDQ 指令

下面展示修改过得地方

1. instr_valid

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

2. need_regids (iaddq有rB)

```
bool need_regids =
    f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                IIRMOVQ, IRMMOVQ, IMRMVQ, IIADDQ };
```

3. need_valC (iaddq有立即数)

```
bool need_valC =
    f_icode in { IIRMOVQ, IRMMOVQ, IMRMVQ, IJXX, ICALL, IIADDQ };
```

4. d_srcB

```
word d_srcB = [
    D_icode in { IOPQ, IRMMOVQ, IMRMVQ, IIADDQ } : D_rB;
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't need register
];
```

5. d_dstE

```
word d_dstE = [
    D_icode in { IRRMOVQ, IIRMOVQ, IOPQ, IIADDQ } : D_rB;
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't write any register
];
```

6. aluA: 把IIADDQ当成“立即数输入”

```
word aluA = [
    E_icode in { IRRMOVQ, IOPQ } : E_valA;
    E_icode in { IIRMOVQ, IRMMOVQ, IMRMVQ, IIADDQ } : E_valC;
    E_icode in { ICALL, IPUSHQ } : -8;
    E_icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
```

7. aluB: 把IIADDQ作为“用valB”那类

```
word aluB = [
    E_icode in { IRMMOVQ, IMRMVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ, IIADDQ } : E_valB;
    E_icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Other instructions don't need ALU
];
```

8. set_cc: 通常iaddq也需要更新条件吗

```
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 };
```

测试

下面进行测试

1. make clean; make VERSION=full

附上截图

pipe-full.hcl x seq-full.hcl

pipe > pipe-full.hcl

```
250
251  ## Select input B to ALU
252  word aluB = [
253      E_icode in { TRMMOVD, TMRMOVD, TOPD, TCALL
```

问题 输出 调试控制台 终端 端口

+ v bash - pipe

xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/CSAPP_Experiment4/pipe\$ make VERSION=full

psim.c:1120:9: warning: 'result' is deprecated: use Tcl_GetStringResult/Tcl_SetResult [-Wdeprecated-declarations]

```
1120 |         fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
      |         ^~~~~~
```

/usr/include/tcl8.6/tcl.h:518:11: note: declared here

```
518 |     char *result TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult")
      |     ^~~~~~
```

psim.c: In function 'report_line':

psim.c:1134:9: warning: 'result' is deprecated: use Tcl_GetStringResult/Tcl_SetResult [-Wdeprecated-declarations]

```
1134 |         fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
      |         ^~~~~~
```

/usr/include/tcl8.6/tcl.h:518:11: note: declared here

```
518 |     char *result TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult")
      |     ^~~~~~
```

psim.c: In function 'report_pc':

psim.c:1190:9: warning: 'result' is deprecated: use Tcl_GetStringResult/Tcl_SetResult [-Wdeprecated-declarations]

```
1190 |         fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
      |         ^~~~~~
```

/usr/include/tcl8.6/tcl.h:518:11: note: declared here

```
518 |     char *result TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult")
      |     ^~~~~~
```

psim.c: In function 'report_state':

psim.c:1204:9: warning: 'result' is deprecated: use Tcl_GetStringResult/Tcl_SetResult [-Wdeprecated-declarations]

```
1204 |         fprintf(stderr, "\tError Message was '%s'\n", sim_interp->result);
      |         ^~~~~~
```

/usr/include/tcl8.6/tcl.h:518:11: note: declared here

```
518 |     char *result TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult")
      |     ^~~~~~
```

./gen-driver.pl -n 4 -f ncopy.js > sdriver.js

../misc/yas sdriver.js

./gen-driver.pl -n 63 -f ncopy.js > ldriver.js

../misc/yas ldriver.js

2. 回归测试:

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

下面附上截图

```
pipe > pipe-full.hcl
251  ## Select input B to ALU
252  word aluB = [
253      E_icode in { IRMMOVQ, IMRMVQ, IOPQ, ICALL,
254      IPUSHQ, IRET, IPOPOQ, IIADDQ } : E_valB;
255      E_icode in { IRRMOVQ, IIRMOVQ } : 0;

xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/CSAPP_Experiment
4/pipe$ (cd ../y86-code; make testpsim)
Makefile:42: 警告: 忽略后缀规则定义的先决条件
Makefile:45: 警告: 忽略后缀规则定义的先决条件
Makefile:48: 警告: 忽略后缀规则定义的先决条件 ...

../pipe/psim -t j-cc.yo > j-cc.pipe
../pipe/psim -t popstest.yo > popstest.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
popstest.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 popstest.pipe pushquestion.pipe pushtest.p
ipe prog1.pipe prog2.pipe prog3.pipe prog4.pipe prog5.pipe prog6.pipe prog7.pipe pro
g8.pipe ret-hazard.pipe

■ (cd ../ptest; make SIM=../pipe/psim)
```

下面附上截图

```
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/CSAPP_Experiment
4/pipe$ (cd ../ptest; make SIM=../pipe/psim)
./optest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 49 ISA Checks Succeed
./jtest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 64 ISA Checks Succeed
./ctest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 600 ISA Checks Succeed
```

3. 专门测试iaddq（现在没修改过），应该是不会出错

```
(cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)
```

下面附上截图

```
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/CSAPP_Experiment
4/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
xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/CSAPP_Experiment
```

以上，说明我们的修改和加入iaddq，是正确的，下面我要修改ncopy.y，将其中可以用iaddq替换的替换掉。

修改ncopy.y

用 iaddq 替换

```
irmovq $1, %r10 + addq %r10, %rax (count++)
```

```
irmovq $1, %r10 + subq %r10, %rdx (len--)
```

```
irmovq $8, %r10 + addq %r10, %rdi/%rsi (指针++)
```

将以上的替换为

```
iaddq $1, %rax
iaddq $-1, %rdx
iaddq $8, %rdi
iaddq $8, %rsi
```

8-way unroll（一次循环处理八个元素）

附上源代码

```
#!/* $begin ncopy-ys */
#####
# ncopy.ys - 8-way unrolled version for Y86 architecture
# Copy a src block of len words to dst.
# Return the number of positive words (>0) contained in src.
#
# Name: 徐文博
# ID: 1320240207
#####

# %rdi = src, %rsi = dst, %rdx = len
ncopy:
#####
# init
    xorq %rax, %rax           # global count = 0
    andq %rdx, %rdx
    jle Done

#####
# 8-way loop
#####
Loop8_Check:
    rrmovq %rdx, %rbx         # rbx = len
    iaddq $-8, %rbx           # rbx = len - 8
    jl Remain                 # if len < 8 -> remainder

Loop8:
    # ---- load 8 ----
    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), %rbx
mrmovq 56(%rdi), %rcx
```

```
# ---- store 8 ----
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 %rbx, 48(%rsi)
rmmovq %rcx, 56(%rsi)
```

```
# ---- local count (reduce rax dependency) ----
xorq %r14, %r14          # local_count = 0
```

```
andq %r8, %r8            # Check r8
jle L0
iaddq $1, %r14
```

L0:

```
andq %r9, %r9            # Check r9
jle L1
iaddq $1, %r14
```

L1:

```
andq %r10, %r10          # Check r10
jle L2
iaddq $1, %r14
```

L2:

```
andq %r11, %r11          # Check r11
jle L3
iaddq $1, %r14
```

L3:

```
andq %r12, %r12          # Check r12
```

```

        jle L4
        iaddq $1, %r14
L4:
        andq %r13, %r13          # Check r13
        jle L5
        iaddq $1, %r14
L5:
        andq %rbx, %rbx          # Check rbx
        jle L6
        iaddq $1, %r14
L6:
        andq %rcx, %rcx          # Check rcx
        jle L7
        iaddq $1, %r14
L7:

        # ---- merge local count ----
        addq %r14, %rax

        # ---- update pointers & len ----
        iaddq $64, %rdi           # src += 8 * 8
        iaddq $64, %rsi           # dst += 8 * 8
        iaddq $-8, %rdx           # len -= 8
        jmp Loop8_Check

#####
# remainder loop (1-way)
#####

Remain:
        andq %rdx, %rdx
        jle Done

Loop1:
        mrmovq 0(%rdi), %r10
        rmmovq %r10, 0(%rsi)

        andq %r10, %r10
        jle R0

```

```
        iaddq $1, %rax
R0:
        iaddq $8, %rdi
        iaddq $8, %rsi
        iaddq $-1, %rdx
        jg Loop1

#####

Done:
        ret

#####

End:
#/* $end ncopy-ys */
```

最后附上截图

```
105         jle R0
106         iaddq $1, %rax
```

问题 输出 调试控制台 终端 端口

xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/0

35	274	7.83
36	286	7.94
37	295	7.97
38	307	8.08
39	316	8.10
40	295	7.38
41	308	7.51
42	320	7.62
43	329	7.65
44	341	7.75
45	350	7.78
46	362	7.87
47	371	7.89
48	350	7.29
49	363	7.41
50	375	7.50
51	384	7.53
52	396	7.62
53	405	7.64
54	417	7.72
55	426	7.75
56	405	7.23
57	418	7.33
58	430	7.41
59	439	7.44
60	451	7.52
61	460	7.54
62	472	7.61
63	481	7.63
64	460	7.19
Average CPE		9.36
Score		22.9/60.0

✧ xwb@xwb-Lenovo-ThinkBook-16p-Gen-4:~/Work/BIT/大三/CSAPP_Experiment/0

