目录

第四章家庭作业、Archlab及PIPE的HCL实现

李一鸣 1160300625

code

github: upupming/CSAPP/chapter4

All solutions and lab reports are in Chinese.

serve locally

clone code

```
git clone https://github.com/upupming/CSAPP.git
cd CSAPP/chapter4
```

install nodejs

```
sudo apt install nodejs nodejs-legacy
```

install `gitbook-cli`

```
npm i -g gitbook-cli
```

install plugins

```
gitbook install
```

serve

```
gitbook serve --no-watch
```

visit link

```
http://localhost:4000
```

generate ebook

prerequisite

- ebook-convert

  ```
  sudo apt install calibre
  ```

generate book

```
gitbook pdf ./ ./CSAPP-chapter4.pdf
gitbook mobi ./ ./CSAPP-chapter4.mobi
gitbook epub ./ ./CSAPP-chapter4.epub
```

ref: gitbook toolchain: ebook

第4章 处理器体系结构 作业

4.46

A. 从练习题4.8的返回值总是0xabcd可推断出在x86-64中pop %rsp将%rsp设置为从内存中读出来的那个值，而不是本身加8后的值。本题中的代码违背了这个规则。 B. 可改成如下代码：

```
addq $8, %rsp
movq -8(%rsp), REG
```

4.47

A. 模仿题中给的数组形式代码，得出指针形式的代码如下：

```
/* Bubble sort: Pointer version */
void bubble_b(long* data, long count){
    long i, last;
    for(last = count-1; last >0; last--){
        for(long* i = data; i < data+last; i++){
            if(*(i+1) < *i){
                /* Swap adjacent elements */
                long t = *(i+1);
                *(i+1) = *(i);
                *(i) = t;
            }
        }
    }
}
```

B. 为得到上述代码的Y86-64程序，先将其编译成X86-64代码：

```
    .file    "bubbleSortUsingPointers.c"
    .text
    .globl    bubble_b
    .type    bubble_b, @function
bubble_b:
.LFB0:
    .cfi_startproc
    subq    $1, %rsi
    jmp    .L2
.L4:
    movq    8(%rax), %rdx
    movq    (%rax), %rcx
    cmpq    %rcx, %rdx
    jge    .L3
    movq    %rcx, 8(%rax)
    movq    %rdx, (%rax)
.L3:
    addq    $8, %rax
    jmp    .L5
.L6:
    movq    %rdi, %rax
.L5:
    leaq    (%rdi,%rsi,8), %rdx
    cmpq    %rdx, %rax
```

```
    jb    .L4
    subq    $1, %rsi
.L2:
    testq    %rsi, %rsi
    jg    .L6
    rep ret
    .cfi_endproc
.LFE0:
    .size    bubble_b, .-bubble_b
    .ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609"
    .section    .note.GNU-stack,"",@progbits
```

模拟上面的x86-64代码，手工编写出下面的Y86-64代码：

```
# void bubble_b(long *data, long count)
# data in %rdi, count in %rsi
bubble_b:
    irmovq    $1, %r8
    subq    %r8, %rsi        # count--
    jmp        outer_judge
inner_loop:
    mrmovq    8(%rax), %rdx    # *(i+1)
    mrmovq    0(%rax), %rcx    # *(i)
    rrmovq    %rdx, %r9
    subq    %rcx, %r9
    jge        inner_incre
    rmmovq    %rcx, 8(%rax)
    rmmovq    %rdx, (%rax)
inner_incre:
    irmovq    $8, %r10
    addq    %r10, %rax
    jmp        inner_judge
outer_loop:
    rrmovq    %rdi, %rax        # i = data
inner_judge:
    addq    %rsi, %rsi        # 2*%rsi
    addq    %rsi, %rsi        # 4*%rsi
    addq    %rsi, %rsi        # 8*%rsi
    addq    %rsi, %rdi
    rrmovq    %rdi, %rdx        # last = last--
    subq    %rax, %rdx
    jg        inner_loop        # if i < data+last, goto inner_loop
    subq    %r8, %rsi        # last--
outer_judge:
    andq     %rsi, %rsi
    jg        outer_loop
    ret
```

4.48

不使用跳转，最多使用三次条件传送，实现冒泡函数6-11行的测试与交换：

```
inner_loop:
    mrmovq    8(%rax), %rdx    # *(i+1)
    mrmovq    0(%rax), %rcx    # *(i)
    rrmovq    %rdx, %r9
    subq    %rcx, %r9
    rmmovq    %rcx, 8(%rax)
```

```
    rmmovq    %rdx, (%rax)
    cmovge    %rdx, 8(%rax)        #关键在此，再次交换回来
    cmovge    %rcx, (%rax)
```

4.50

switch的跳转表实现：Y86-64指令集不包含间接跳转指令（jmp *.L4(,%rsi,8)），可以把计算好的地址入栈，再执行ret指令。
首先根据题中所给的C代码编译出x86-64汇编代码：

```
.file    "450.c"
    .text
    .globl    switchv
    .type    switchv, @function
switchv:
.LFB23:
    .cfi_startproc
    cmpq    $2, %rdi
    je    .L7
    cmpq    $2, %rdi
    jg    .L4
    testq    %rdi, %rdi
    je    .L5
    jmp    .L2
.L4:
    cmpq    $3, %rdi
    je    .L6
    cmpq    $5, %rdi
    je    .L7
    jmp    .L2
.L5:
    movl    $2730, %eax
    ret
.L6:
    movl    $3276, %eax
    ret
.L2:
    movl    $3549, %eax
    ret
.L7:
    movl    $3003, %eax
    ret
    .cfi_endproc
.LFE23:
    .size    switchv, .-switchv
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string    "idx = %ld, val = 0x%lx\n"
    .text
    .globl    main
    .type    main, @function
main:
.LFB24:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    pushq    %rbx
    .cfi_def_cfa_offset 24
    .cfi_offset 3, -24
```

```
    subq    $8, %rsp
    .cfi_def_cfa_offset 32
    movl    $0, %ebx
    jmp    .L9
.L10:
    leaq    -1(%rbx), %rbp
    movq    %rbp, %rdi
    call    switchv
    movq    %rax, %rcx
    movq    %rbp, %rdx
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
    addq    $1, %rbx
.L9:
    cmpq    $7, %rbx
    jle    .L10
    movl    $0, %eax
    addq    $8, %rsp
    .cfi_def_cfa_offset 24
    popq    %rbx
    .cfi_def_cfa_offset 16
    popq    %rbp
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE24:
    .size    main, .-main
    .ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609"
    .section    .note.GNU-stack,"",@progbits
```

在x86-64代码基础上，手工编写Y86-64的代码：

```
       .pos    0
switchv:
    rrmovq    %rdi, %r8    # 0x000, +2
    irmovq    $2, %r15    # 0x002, +a
    subq    %r15, %r8        # 0x00c, +2
    mrmovq    $32(jt), %r8# case_2_5    0x00e, +a
    mrmovq    (jt), %r9    # case_3_5    0x018, +a
    cmove    %r8, %r11    # if idx==2, return to case_2_5        0x22, +2
    cmovg    %r9, %r11    # if idx>2, return to case_3_5        0x24, +2
    andq    %rdi, %rdi    #                                    0x26, +2
    mrmovq    $8(jt), %r10    # 0x28, +a
    cmove    %r10, %r11    # if idx==0, return to case_0    # 0x32, +2
    push    %r11        #    0x34, +2
    ret                    #    0x36, +1
    mrmovq    $24(jt), %r11# else, return to case_default        0x37, +a
    push    %r11        #    0x41, +2
    ret                    #    0x43, +1
case_3_5:
    rrmovq    %rdi, %r8    #    0x44, +2
    irmovq    $3, %r16    #    0x46, +a
    subq    %r16, %r8    #    0x50, +2
    mrmovq    $16(jt), %r8#    0x52, +a
    cmove    %r8, %r11    # if idx==3, return to case_3    0x5c, +2
    rrmovq    %rdi, %r8    # 0x5e, +2
    subq    $5, %r8        # 0x60, +2
    mrmovq    $32(jt), %r9# 0x62, +10
```

```
    cmove    %r9, %r11    # if idx==5, return to case_2_5    0x6c, +2
    push    %r11        # 0x6e, +2
    ret                    # 0x70, +1
    mrmovq    $24(jt), %r11# else return to case_default    0x71, +a
    push    %r11        #    0x7b, +2
    ret                    #    0x7d, +1
case_0:
    irmovq    $2730, %rax    #    0x7e, +a
    ret                    #    0x88, +1
case_3:
    irmovq    $3276, %rax    #    0x89, +a
    ret                    #    0x93, +1
case_default:
    irmovq    $3549, %rax    #    0x94, +a
    ret                    #    0x9e, +1
case_2_5:
    irmovq    $3003, %rax    #    0x9f, +a
    ret                    #    0xa9, +1

jt:
    .quad 0x0001000100000000    # case_3_5(01000100)    0xaa, +8
    .quad 0x1110110100000000    # case_0(01111011)    0xb2, +8
    .quad 0x0110001000000000    # case_3(10001001)    0xba, +8
    .quad 0x0001011000000000    # case_default(10010100)    0xc2, +8
    .quad 0x1111011000000000    # case_2_5(10011111)        0xca, +8
```

4.51

只把iaadq的各阶段计算写一下： |阶段(Stage)| 指令iaddq V, rB | |---|---| |取指(Fetch)| icode:ifun ← M1[PC]
rA: rB ← M1[PC+1]
valC ← M8[PC+2]
valP ← PC+10 | |译码(Decode)| valA ← R[rB]| |执行(Execute) |valE ← valA+valC| |访存(Memory) || |写回(Write Back)| R[rB] ← valE| |更新PC(PC Update)| PC ← valP|

4.52

SEQ的iaddq指令见实验报告《Archlab》中的Part B。

4.54

PIPE的iaddq已在实验《Archlab》Part C中实现。

4.56

要求反向选择(backward taken)、正向不选择(forward not-taken)。
这种策略的成功率大约为65%。
`pipe-btfnnt.hcl` 声明如下：

```
##### Jump conditions referenced explicitly
wordsig UNCOND 'C_YES'                # Unconditional transfer
```

先将 `pipe-btfnnt.hcl` 重命名为 `original-pipe-btfnnt.hcl`

依题意对 `pipe-btfnnt.hcl` 有如下修改：

- `f_pc` 的修改：

```
word f_pc = [
    # Mispredicted branch.  Fetch at incremented PC
    # 1. backward taken error
    M_icode == IJXX && M_ifun != UNCOND && M_valE < M_valA && !M_Cnd : M_valA;

    # 2. forward not-taken error
+    M_icode == IJXX && M_ifun != UNCOND && M_valE >= M_valA && M_Cnd : M_valE;

    # Completion of RET instruction
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];
```

- `f_predPC` 的修改：

```
# Predict next value of PC
word f_predPC = [
    # BBTFNT: This is where you'll change the branch prediction rule
    # 3. 后向选择
    f_icode == IJXX && f_ifun != UNCOND && f_valC < f_valP : f_valC;
    # 4. 前向不选择
    f_icode == IJXX && f_ifun != UNCOND && f_valC >= f_valP : f_valP;
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];
```

- `aluA` 的修改：

```
# BBTFNT: When some branches are predicted as not-taken, you need some
# way to get valC into pipeline register M, so that
# you can correct for a mispredicted branch.

## pass valC by M_valE, pass valP by M_valA
## Select input A to ALU
word aluA = [
    E_icode in { IRRMOVQ, IOPQ } : E_valA;
    E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
    E_icode in { ICALL, IPUSHQ } : -8;
    E_icode in { IRET, IPOPQ } : 8;
    # 5. 以便从错误中恢复
    E_icode in { IJXX } : E_valC;
    # Other instructions don't need ALU
];
```

- `D_bubble` 的修改：

```
bool D_bubble =
    # Mispredicted branch
    # (E_icode == IJXX && !e_Cnd) ||
    # 6. 修改
    (E_icode == IJXX && E_ifun != UNCOND && E_valC < E_valA && !e_Cnd) ||
    (E_icode == IJXX && E_ifun != UNCOND && E_valC >= E_valA && e_Cnd)  ||
    # BBTFNT: This condition will change
    # 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 };
```

- `E_bubble` 的修改：

```
bool E_bubble =
    # Mispredicted branch
    # (E_icode == IJXX && !e_Cnd) ||
    # 7. 修改
    # backward taken error or forward not-taken error
    (
    (E_icode == IJXX && E_ifun != UNCOND && E_valC < E_valA && !e_Cnd) ||
    (E_icode == IJXX && E_ifun != UNCOND && E_valC >= E_valA && e_Cnd)
    ) ||
    # BBTFNT: This condition will change
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } &&
     E_dstM in { d_srcA, d_srcB};
```

修改后的文件在这里，实测《Archlab》中所描述的测试通过。

4.58

有以下几个关键点：

1. popq两次取指
2. 第一次取popq做iaddq $8, %rsp，第二次取popq2做mrmovq -8(%rsp), rA
3. load/use发生在mrmovq时

阶段	popq rA	popq2 rA
实际执行	iaddq $8, %rsp	mrmovq -8(%rsp), rA
F	valP = PC	valP = PC + 2
D	valB=R[%rsp]	valB=R[%rsp]
E	valE=valB+8	valE=valB-8
M		valM=M8[valE]
W	R[rsp]=valE	R[rA]=valM

`pipe-1w.hcl` 修改如下(修改好后执行diff所得)：

```
--- origin-pipe-1w.hcl    2017-12-02 00:49:15.000000000 -0800
+++ pipe-1w.hcl    2017-12-02 00:49:15.000000000 -0800
@@ -157,6 +157,7 @@
 ## so that it will be IPOP2 when fetched for second time.
 word f_icode = [
     imem_error : INOP;
+    D_icode == IPOPQ : IPOP2;
     1: imem_icode;
 ];

@@ -169,7 +170,7 @@
 # Is instruction valid?
 bool instr_valid = f_icode in
     { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
-      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
+      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IPOP2 };

 # Determine status code for fetched instruction
```

```
 word f_stat = [
@@ -182,7 +183,7 @@
 # Does fetched instruction require a regid byte?
 bool need_regids =
     f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
-             IIRMOVQ, IRMMOVQ, IMRMOVQ };
+             IIRMOVQ, IRMMOVQ, IMRMOVQ, IPOP2 };

 # Does fetched instruction require a constant word?
 bool need_valC =
@@ -192,6 +193,7 @@
 word f_predPC = [
     f_icode in { IJXX, ICALL } : f_valC;
     ## 1W: Want to refetch popq one time
+    f_icode == IPOPQ : f_pc;
     1 : f_valP;
 ];

@@ -204,14 +206,14 @@
 ## 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;
+    D_icode in { 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, IMRMOVQ  } : D_rB;
-    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
+    D_icode in { IPUSHQ, IPOPQ, IPOP2, ICALL, IRET } : RRSP;
     1 : RNONE;  # Don't need register
 ];

@@ -224,7 +226,7 @@
 ## What register should be used as the M destination?
 word d_dstM = [
-    D_icode in { IMRMOVQ, IPOPQ } : D_rA;
+    D_icode in { IMRMOVQ, IPOP2 } : D_rA;
     1 : RNONE;  # Don't write any register
 ];

@@ -255,7 +257,7 @@
 word aluA = [
     E_icode in { IRRMOVQ, IOPQ } : E_valA;
     E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
-    E_icode in { ICALL, IPUSHQ } : -8;
+    E_icode in { ICALL, IPUSHQ, IPOP2 } : -8;
     E_icode in { IRET, IPOPQ } : 8;
     # Other instructions don't need ALU
 ];
@@ -263,7 +265,7 @@
 ## Select input B to ALU
 word aluB = [
     E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
-             IPUSHQ, IRET, IPOPQ } : E_valB;
+             IPUSHQ, IRET, IPOPQ, IPOP2 } : E_valB;
     E_icode in { IRRMOVQ, IIRMOVQ } : 0;
```

```
      # Other instructions don't need ALU
 ];
@@ -292,13 +294,13 @@

 ## Select memory address
 word mem_addr = [
-    M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
-    M_icode in { IPOPQ, IRET } : M_valA;
+    M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ, IPOP2 } : M_valE;
+    M_icode in { IRET } : M_valA;
      # Other instructions don't need address
 ];

 ## Set read control signal
-bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };
+bool mem_read = M_icode in { IMRMOVQ, IRET, IPOP2 };

 ## Set write control signal
 bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };
@@ -350,7 +352,7 @@
 bool F_bubble = 0;
 bool F_stall =
      # Conditions for a load/use hazard
-    E_icode in { IMRMOVQ, IPOPQ } &&
+    E_icode in { IMRMOVQ, IPOP2 } &&
       E_dstM in { d_srcA, d_srcB } ||
      # Stalling at fetch while ret passes through pipeline
      IRET in { D_icode, E_icode, M_icode };
@@ -359,7 +361,7 @@
 # At most one of these can be true.
 bool D_stall =
      # Conditions for a load/use hazard
-    E_icode in { IMRMOVQ, IPOPQ } &&
+    E_icode in { IMRMOVQ, IPOP2 } &&
       E_dstM in { d_srcA, d_srcB };

 bool D_bubble =
@@ -367,7 +369,7 @@
      (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 }) &&
+    !(E_icode in { IMRMOVQ, IPOP2 } && E_dstM in { d_srcA, d_srcB }) &&
      # 1W: This condition will change
        IRET in { D_icode, E_icode, M_icode };

@@ -378,7 +380,7 @@
      # Mispredicted branch
      (E_icode == IJXX && !e_Cnd) ||
      # Conditions for a load/use hazard
-    E_icode in { IMRMOVQ, IPOPQ } &&
+    E_icode in { IMRMOVQ, IPOP2 } &&
       E_dstM in { d_srcA, d_srcB};

 # Should I stall or inject a bubble into Pipeline Register M?
```

ArchLab

姓名：李一鸣　　学号：1160300625

Handout Instructions

根据sim下的README文档， 对Makefile进行修改。
取消Makefile中GUI版的注释：

```
GUIMODE=-DHAS_GUI
```

在ubuntu下，运行命令找到对应的TKLIBS：

```
1160300625@ubuntu:/usr/lib/x86_64-linux-gnu$ ls | grep libtcl
```

输出是：

```
libtcl8.6.so
libtcl8.6.so.0
```

因此可以将Makefile中的TKLIBS做相应修改：

```
TKLIBS=-L/usr/lib/x86_64-linux-gnu -ltk -ltcl
```

接着安装Tcl、Tk开发文件：

```
$ sudo apt install tcl8.6-dev tk8.6-dev
```

同样，经过查询资料与命令验证：

```
1160300625@ubuntu:/usr/include/tcl8.6$ ls | grep tcl.h
```

输出为：

```
tcl.h
```

Makefile做相应修改：

```
TKINC=-isystem /usr/include/tcl8.6
```

经过我的反复测试，还需要安装flex、bison：

```
$ sudo apt install flex bison
```

修改好后，执行下面的命令：

```
make clean; make
```

报出如下错误：

```
psim.c:853:8: error: ‘Tcl_Interp {aka struct Tcl_Interp}’ has no member named ‘result’ interp->result =
"No arguments allowed";
```

Tcl_Interp很明显是在tcl.h中定义的，怎么会没有result成员呢？于是看看tcl.h中到底怎样写的：

```
$ vi /usr/include/tcl8.6/tcl.h
```

找到Tcl_Interp的定义如下：

```
typedef struct Tcl_Interp
#ifndef TCL_NO_DEPRECATED
{
    /* TIP #330: Strongly discourage extensions from using the string
     * result. */
#ifdef USE_INTERP_RESULT
```

```
    char *result TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult");
                                /* If the last command returned a string
                                 * result, this points to it. */
    void (*freeProc) (char *blockPtr)
            TCL_DEPRECATED_API("use Tcl_GetStringResult/Tcl_SetResult");
                                /* Zero means the string result is statically
                                 * allocated. TCL_DYNAMIC means it was
                                 * allocated with ckalloc and should be freed
                                 * with ckfree. Other values give the address
                                 * of function to invoke to free the result.
                                 * Tcl_Eval must free it before executing next
                                 * command. */
#else
    char *resultDontUse; /* Don't use in extensions! */
    void (*freeProcDontUse) (char *); /* Don't use in extensions! */
#endif
#ifdef USE_INTERP_ERRORLINE
    int errorLine TCL_DEPRECATED_API("use Tcl_GetErrorLine/Tcl_SetErrorLine");
                                /* When TCL_ERROR is returned, this gives the
                                 * line number within the command where the
                                 * error occurred (1 if first line). */
#else
    int errorLineDontUse; /* Don't use in extensions! */
#endif
}
#endif /* TCL_NO_DEPRECATED */
Tcl_Interp;
```

可以看出这是tcl8.6优化的缘故，可能会定义result或者resultDontUse两者之一，因此不一定有result成员，故考虑使用tcl8.5。

```
sudo apt install tcl8.5-dev tk8.5-dev
```

同样需要修改Makefile：

```
TKINC=-isystem /usr/include/tcl8.5
```

不幸的是， 还有错误：

```
Building the pipe-std.hcl version of PIPE
../misc/hcl2c -n pipe-std.hcl < pipe-std.hcl > pipe-std.c
gcc -Wall -O2 -isystem /usr/include/tcl8.5 -I../misc -DHAS_GUI -o psim psim.c pipe-std.c \
../misc/isa.c -L /usr/lib/x86_64-linux-gnu -ltk -ltcl -lm
/usr/bin/ld: cannot find -ltk
/usr/bin/ld: cannot find -ltcl
```

查阅资料得知，这是参数错误引起的。也就是说-ltk、-ltcl不是合法参数，对Makefile做如下修改：

```
TKLIBS=-L /usr/lib/x86_64-linux-gnu -ltk8.5 -ltcl8.5
```

最终执行：

```
make clean; make
```

这下终于make成功了。

Part A

先用下面的命令得到C代码的x86-64汇编代码：

```
1160300625@ubuntu:~/hitcis/lab5/sim/misc$ gcc -Og -S examples.c
```

得到的汇编代码在这里。

A-1 sum.ys: Iteratively sum linked list elements(迭代求和)

根据x86-64代码，很容易得出Y86-64代码。
在上述代码中根据题目要求编写了main函数对函数进行测试。
下面用YAS(Y86-64 Assember)汇编器进行汇编：

```
~/hitcis/lab5/sim/misc$ ./yas sum.ys
```

再用YIS(Y86-64 instruction set simulator)指令集模拟器模拟程序的执行：

```
~/hitcis/lab5/sim/misc$ ./yis sum.yo
```

结果如下：

```
Stopped in 26 steps at PC = 0x13. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000cba
%rsp: 0x0000000000000000 0x0000000000000200
%r8: 0x0000000000000000 0x0000000000000c00

Changes to memory:
0x01f0: 0x0000000000000000 0x000000000000005b
0x01f8: 0x0000000000000000 0x0000000000000013
```

可以看到没有任何error并且最终%rax的值为0xcba，结果正确。

A-2 rsum.ys: Recursively sum linked list elements(递归求和)

同样根据x86-64代码，很容易得到Y86-64代码。 测试：

```
$ ./yas rsum.ys
$ ./yis rsum.yo
```

结果如下：

```
Stopped in 37 steps at PC = 0x13. Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000cba
%rsp: 0x0000000000000000 0x0000000000000200
%r8: 0x0000000000000000 0x000000000000000a

Changes to memory:
0x01c0: 0x0000000000000000 0x0000000000000086
0x01c8: 0x0000000000000000 0x0000000000000c00
0x01d0: 0x0000000000000000 0x0000000000000086
0x01d8: 0x0000000000000000 0x00000000000000b0
0x01e0: 0x0000000000000000 0x0000000000000086
0x01e8: 0x0000000000000000 0x000000000000000a
0x01f0: 0x0000000000000000 0x000000000000005b
0x01f8: 0x0000000000000000 0x0000000000000013
```

可以看到，%rax最终为0xcba，结果正确。

A-3 copy.ys: Copy a source block to a destination block

根据x86-64代码和体中所给的测试用例，写出Y86-64代码。
运行：

```
./yas copy.ys
./yis copy.yo
```

输出结果为：

```
Stopped in 43 steps at PC = 0x13. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000cba
%rcx: 0x0000000000000000 0x0000000000000c00
%rsp: 0x0000000000000000 0x0000000000000200
%rsi: 0x0000000000000000 0x0000000000000048
%rdi: 0x0000000000000000 0x0000000000000030
%r8: 0x0000000000000000 0x0000000000000008

Changes to memory:
0x0030: 0x0000000000000111 0x000000000000000a
0x0038: 0x0000000000000222 0x00000000000000b0
0x0040: 0x0000000000000333 0x0000000000000c00
0x01f0: 0x0000000000000000 0x000000000000006f
0x01f8: 0x0000000000000000 0x0000000000000013
```

可以看到，dest内存部分已经被修改成了与src相同的值。

Part B

题目要求修改sim/seq下的 `seq-full.hcl` 来实现课本中的iaddq指令，并在前面注释中写上自己的学号、姓名以及iaddq的计算过程。

B-1 修改 `seq-full.hcl` 以加入iaddq指令

在 `seq-full.hcl` 中有下面的定义：

```
# Instruction code for iaddq instruction
wordsig IIADDQ 'I_IADDQ'
```

在其头文件 `isa.h` 中又有 `I_IADDQ` 的定义：

```
/ Different instruction types /
typedef enum { I_HALT, I_NOP, I_RRMOVQ, I_IRMOVQ, I_RMMOVQ, I_MRMOVQ,
I_ALU, I_JMP, I_CALL, I_RET, I_PUSHQ, I_POPQ,
I_IADDQ, I_POP2 } itype_t;
```

可见 `I_IADDQ` 被定义为十六进制值C。

接下来根据家庭作业4.51的结果对各阶段进行修改，先把4.51的答案再重复一遍：

阶段(Stage)	指令iaddq V, rB
取指(Fetch)	icode:ifun ← M1[PC] rA: rB ← M1[PC+1] valC ← M8[PC+2] valP ← PC+10
译码(Decode)	valA ← R[rB]
执行(Execute)	valE ← valA+valC
访存(Memory)	
写回(Write Back)	R[rB] ← valE
更新PC(PC Update)	PC ← valP

1. 取指(Fetch)
2. `icode` 不需修改
3. `ifun` 不需修改
4. `instr_valid` 需要加入 `IIADDQ`

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

5. iaddq指令包含寄存器指示符字节， `need_regids` 需要加入'IIADDQ'(与上面类似)
6. iaddq指令包含常数字， `need_valC` 需要加入'IIADDQ'
7. 译码(Decode)和写回(Write Back)
8. 读端口A的地址连接被设置为 `rB` ，对 `srcA` 修改：

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

9. `srcB` 不需修改
10. 在写回阶段 `valE` 写到了 `rB` ，对destE修改如下：

```
## 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
];
```

11. `destM` 由于后续没有从内存读数据到寄存器，不需修改
12. 执行(Execute)
13. iaddq指令中 `valC` 作为ALU(算数逻辑单元)的 `aluA` ，因此对 `aluA` 修改如下：

```
## 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
];
```

14. iaddq指令中 `valA` 作为ALU(算数逻辑单元)的 `aluB` ，因此对 `aluB` 修改如下：

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

15. iaddq仍然是执行加法指令， `alufun` 不需修改

16. iaddq需要设置条件码，因此 `set_cc` 修改如下：

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

17. 访存(Memory)
iaddq指令只是对立即数和寄存器进行操作，不需要读写内存，因此这部分不许做任何改变。

B-2 生成SEQ并测试

修改完之后，就要根据修改的HCL文件生成一个新的SEQ模拟器(ssim)并进行测试。
生成SEQ模拟器(注意：此处可能同样需要修改Makefile，请参照Handout Instruction)：

> ~/hitcis/lab5/sim/seq$ make VERSION=full

用Y86-64程序验证模拟器：

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

输出结果如下：

```
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
```

为验证其正确性，用YIS运行一下asumi.yo：

```
$ ./../misc/yis ../y86-code/asumi.yo
```

输出结果如下：

```
Stopped in 32 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:    0x0000000000000000    0x0000abcdabcdabcd
%rsp:    0x0000000000000000    0x0000000000000100
%rdi:    0x0000000000000000    0x0000000000000038
%r10:    0x0000000000000000    0x0000a000a000a000

Changes to memory:
0x00f0:    0x0000000000000000    0x0000000000000055
0x00f8:    0x0000000000000000    0x0000000000000013
```

可见两者结果完全相同，因此自己修改后的SEQ成功实现了iaddq指令，但是还需进一步验证。

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

注：圆括号表示不要真正切换目录。
输出如下：

```
../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.seq
 prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq prog8.seq ret-hazard.seq
```

可见y86-code下的程序测试都通过了，这只证明新的SEQ没有使原有的指令发生错误，还需进一步验证：

验证 `leave` ：

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

输出如下：

```
./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
```

验证 `iaddq` ：

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

输出如下：

```
./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
```

大功告成!

Part C

在 sim/pipe目录下，有下列文件：

- `nocopy.c` ：C代码程序
- `nocopy.ys` ：用非流水线编写的Y86-64代码
- `pipe-full.hcl` ：加入了IIADDQ常量定义的PIPE的HCL实现代码

要求是修改 `nocopy.ys` 和 `pipe-full.hcl` 以使程序运行尽量快，同时还有下列约束：

- `nocopy.ys` 要对任意任意类型的数组都能起作用。
- `nocopy.ys` 要能复制src到dest，并且返回%rax为正确的计数。
- `nocopy.ys` 汇编产生的 `nocopy.yo` 不能超过1000字节，用下面的脚本验证：

  ```
  unix> ./check-len.pl < ncopy.yo
  ```

- `pipe-full.hcl` 必须通过 `../y86-code` 和 `../ptest` (不需用'-i'测试iaddq指令)中的测试。
- 如果感兴趣，可以实现一下iaddq指令。

读到这里，再看看 `nocopy.ys` ，其中有多处先用 `irmovq` 、再用 `addq` 来实现加立即数。
我就想，想要快那就必须加入 `iaddq` 指令。

C-1 修改 **`pipe-full.hcl`** 以加入iaddq指令

1. 取指(Fetch)
2. PC的选择 `f_pc` 不需修改
3. 用来确定取指阶段icode:ifun的 `f_icode` 、 `f_ifun` 不需修改
4. `instr_valid` 加入 `IIADDQ`
5. 用来确定取指指令的状态码的 `f_stat` 不需修改
6. `need_regids` 需要加入 `IIADDQ`
7. iaddq指令有常数字， `need_valC` 需要加入 `IIADDQ`
8. iaddq指令PC预测逻辑仍然选择valP， `f_predPC` 不需修改
9. 译码(Decode)和写回(Write Back)
10. 读端口A的地址连接被设置为 `rB` ，对 `srcA` 的修改类似Part B
11. `d_srcB` 不需修改
12. 在写回阶段 `valE` 写到了 `rB` ，对 `d_dstE` 的修改类似Part B
13. `d_destM` 由于后续没有从内存读数据到寄存器，不需修改
14. `d_valA` 用来将 `valP` 合并到 `valA` ，并且实现数据转发，不需修改
15. `d_valB` 用来实现数据转发，不需修改
16. 执行(Execute)
17. iaddq指令中 `valC` 作为ALU(算数逻辑单元)的 `aluA` ， `aluA` 的修改类似Part B
18. iaddq指令中 `valA` 作为ALU(算数逻辑单元)的 `aluB` ， `aluB` 的修改类似Part B
19. iaddq指令仍然执行加法指令， `alufun` 不需修改
20. iaddq指令在不是非法指令情况下，需要更新条件码， `set_cc` 修改如下：

    ```
    ## Should the condition codes be updated?
      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 };
    ```

21. `e_valA` 直接传递不需修改
22. `e_icode` 在条件传递无误时，用 `E_dstE` 赋值，否则置为 `RNONE` ，不需修改
23. 访存(Memory)
    iaddq指令只是对立即数和寄存器进行操作，不需要读写内存，因此这部分不许做任何改变。
24. 流水线寄存器(Pipeline Register Control) iaddq没有影响到气泡的处理，这部分不需修改

C-2 生成PIPE并测试

运行下面的命令(确保Makefile修改正确，参见Part A)：

```
~/hitcis/lab5/sim/pipe$ make VERSION=full
```

输出如下：

```
# Building the pipe-full.hcl version of PIPE
../misc/hcl2c -n pipe-full.hcl < pipe-full.hcl > pipe-full.c
gcc -Wall -O2 -isystem /usr/include/tcl8.5 -I../misc -DHAS_GUI -o psim psim.c pipe-full.c \
    ../misc/isa.c -L/usr/lib -ltk8.5 -ltcl8.5 -lm
./gen-driver.pl -n 4 -f ncopy.ys > sdriver.ys
../misc/yas sdriver.ys
./gen-driver.pl -n 63 -f ncopy.ys > ldriver.ys
../misc/yas ldriver.ys
```

运行 `asumi.yo` 验证PIPE：

```
~/hitcis/lab5/sim/pipe$ ./psim -t ../y86-code/asumi.yo > out_my.txt
```

将输出重定向至out_my.txt。
可见通过了ISA校对。

进一步运行 `sim/y86-code` 下的程序进行验证：

```
~/hitcis/lab5/sim/pipe$ (cd ../y86-code; make testpsim)
```

▶ 输出如下(点击展开)：

```
../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 p
rog2.pipe prog3.pipe prog4.pipe prog5.pipe prog6.pipe prog7.pipe prog8.pipe ret-hazard.pipe
```

</details> 同样没有任何错误。 进一步运行 `sim/ptest` 下的程序测试 `leave` 指令：

> ~/hitcis/lab5/sim/pipe$ (cd ../ptest; make SIM=../pipe/psim)

输出如下：

```
./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
```

同样ISA验证通过。 进一步用'-i'测试 `iaddq` 指令：

> ~/hitcis/lab5/sim/pipe$ (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)

输出如下：

```
./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
```

ISA验证通过，至此PIPE的修改已经完成了测试。下一步就是修改 `nocopy.ys` 了。

加入 `iaddq` 修改、对循环展开以优化 `nocopy.ys`

修改后的代码在这里。
其主要思想是分治，写这个代码需要不断修改，大概花了我一天的时间。
举个例子，要复制127个元素的数组，程序会这样运行：

- 按照8个元素一份复制15次，共复制120个元素
- 剩下7个元素4个一份复制1次，共复制4个元素
- 剩下3个元素2个一份复制一次，共复制2个元素
- 剩余1个元素直接复制即可。

</details> 汇编 `ncopy.ys` ：

> ~/hitcis/lab5/sim/pipe$ ./../misc/yas ncopy.ys

检验字节数：

> $ ./check-len.pl < ncopy.yo

输出为：

> ncopy length = 993 bytes

符合不超过1000字节的要求。

利用 `gen-driver.pl` 生成测试程序：

```
$ make drivers
```

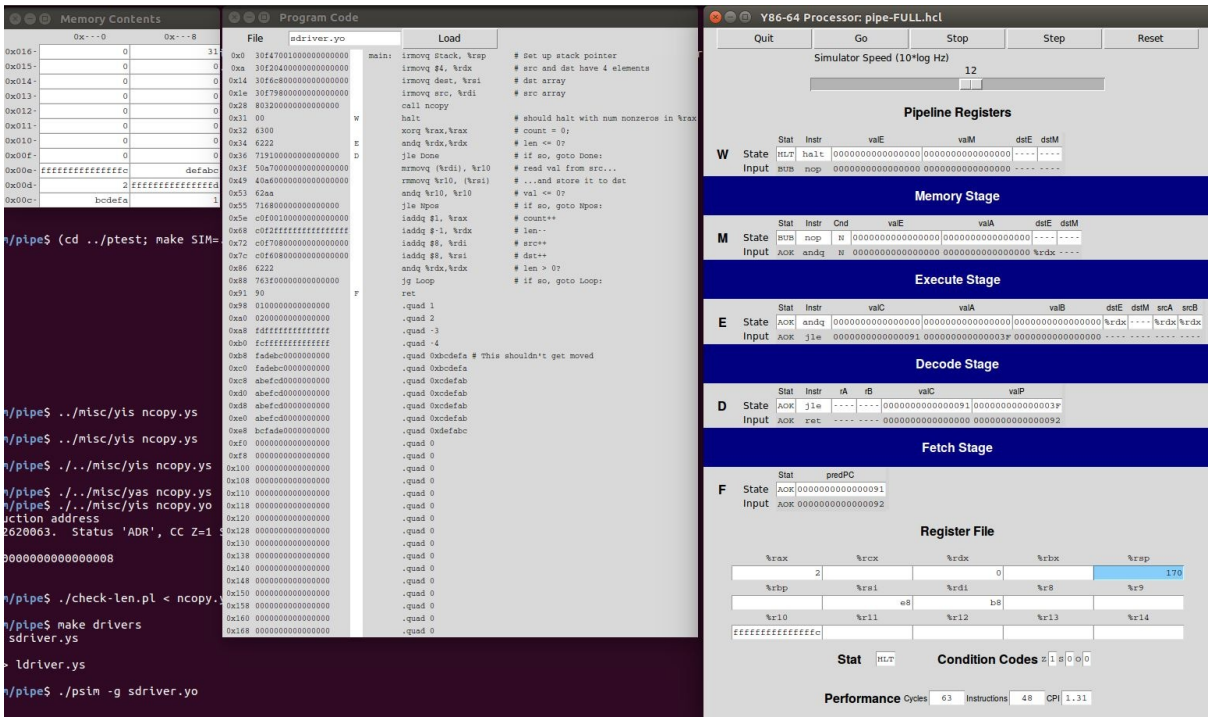
输出为：

```
./gen-driver.pl -n 4 -f ncopy.ys > sdriver.ys
../misc/yas sdriver.ys
./gen-driver.pl -n 63 -f ncopy.ys > ldriver.ys
../misc/yas ldriver.ys
```

用4个元素的数组测试：

```
$ ./psim -g sdriver.yo
```

结果如下图所示：



若直接用tty模式：

```
$ ./psim -t sdriver.yo
```

这条指令会输出运行的详细信息，很详细，最终没有出错。

最后用63个元素的数组进行测试：

```
$ ./psim -t ldriver.yo > ldriver.log
```

输出的log足足有9282行，最后两行为：

```
ISA Check Succeeds
CPI: 740 cycles/608 instructions = 1.22
```

因此测试通过。

为了用YIS验证 `ncopy.ys` ，运行下面的命令：

```
~/hitcis/lab5/sim/pipe$ ../misc/yis sdriver.yo
```

输出如下：

```
Stopped in 48 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    0x00000000000000b8
%r10:    0x0000000000000000    0x0000000000000004

Changes to memory:
0x00c8:    0x0000000000cdefab    0xffffffffffffffff
0x00d0:    0x0000000000cdefab    0xfffffffffffffffe
0x00d8:    0x0000000000cdefab    0x0000000000000003
0x00e0:    0x0000000000cdefab    0x0000000000000004
0x0168:    0x0000000000000000    0x0000000000000031
```

运行通过，成功复制了内存，并将%rax设置成了正确值。

进一步变化数组长度，用YIS进行测试：

```
$ ./correctness.pl
```

▶ Details

输出为(点击展开)：

```
Simulating with instruction set simulator yis
    ncopy
0    OK
1    OK
2    OK
3    OK
4    OK
5    OK
6    OK
7    OK
8    OK
9    OK
10    OK
11    OK
12    OK
13    OK
14    OK
15    OK
16    OK
17    OK
18    OK
19    OK
20    OK
21    OK
22    OK
23    OK
24    OK
25    OK
26    OK
27    OK
28    OK
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
```

</details> 测试通过。

用PIPE进行测试：

```
$ ./correctness.pl -p
```

输出结果同上，测试通过。

最后看一下CPE(circles per element)：

```
$ ./benchmark.pl > score
```

输出结果重定向至score结果，在这里。
这个得分还算可以，理论上 `ncopy.ys` 可以进一步优化。