

Project 1: Optimizing the Performance of a Pipelined Processor

516021910127, Yifeng Gao, xdek@msn.com
515021910302, Chacha Chen, chachachen1997@gmail.com
516021910495, Zhixin Ling, 1069066484@qq.com, group leader

April 29, 2018

1 Introduction

Throughout this lab, we learned about the design and implementation of a pipelined Y86 processor, optimizing both it and a benchmark program to maximize performance. After this lab, not only did we all have a keen appreciation for the interactions between code and hardware that affect the performance of your programs, but also became more familiar with linux operations and computer architecture etc.

In part A, we implemented 3 basic functions using assembly language and in part B, we implemented new instructions `iaddl` and `leave` in the sequence processor and apply `iaddl` to our Y86 assembly programs. In part C, we implemented new instructions `iaddl` and `leave` in the pipeline processor and optimized a program with some optimization techniques and `iaddl` instruction.

Arrangement: Zhixin Ling, as the group leader, finished Part C and the corresponding report section. Chacha Chen and Yifeng Gao work cooperatively on Part A and Part B as well as the corresponding report section and the integration of the report and final hand-in directories.

2 Experiments

The experiment includes 3 parts.

2.1 Part A

2.1.1 Analysis

- **sum.y** is a program that iteratively sums the elements of a linked list.

The basic idea is that we use a conditional jump in a loop which iteratively check whether the next element is equal to zero and if not add up the value to the sum.

- In init part, the stack structure is set up, then the program jumps to Main function, and finally halts.
 - In Main, we first store the first element to the stack before a call to function `sum_list`.
 - In `sum_list` function, we first do the conventions which saves a copy of initial `%ebp` and set `%ebp` to the beginning of the stack frame. Then we initialize the `sum=0`, and then go to a loop which iteratively add up elements' value into our sum.
 - In loop: firstly, the element pointed to is added and then, we increment the pointer address which make it points to the next element. If the next element is equal to zero, jump to done, otherwise loop again.
 - In done: we resume the `%esp` and `%ebp` to the initial value set in init part. Then we can safely let Main function return.
- **rsum.js** is a program that recursively sums the elements of a linked list. This most of the code is similar to the code in `sum.js`, except that it should use a function `rsum_list` that recursively sums a list of numbers.
 - In `rsum_list`, the key idea is that we use `%eax` to store the iterative temporary sum meanwhile store the value of the current element in `%edx`. Also, a very important point is that we should store the address of the next element(if it is not zero) always in `8(%ebp)`, such that in every recursive step, we always update the desired element, which in this case we update `element[i + 1]` with the sum of all elements from $i + 1$ to the end.
 - **copy.js** copies a block of words from one part of memory to another (non-overlapping area) area of memory, computing the checksum (Xor) of all the words copied.
 - The initialization step is similar to the above implementations.
 - In Main: firstly, the store the `src`, `dest` and `len` into main function stack frame for future use. After these preliminaries, `copy_block` function is called. After returning from `copy_block`, we need to resume the `esp` and `ebp` to the initial value set in init part and this is done by "done" function part as similar to above implementations. Finally Main function is returned.
 - `copy_block`: In `copy_block`, firstly we do the conventions like saving a copy of caller's `ebp` and set `ebp` to the beginning of `copy_block`'s stack frame. Then we use 3 registers `%ebx` `%ecx`, `%esi` to store temporary needed values for iteration. Also, `%eax`, the stored length, is subtracted by 1 and a conditional jump instruction was added to terminated the loop when the length is equal to zero. In each

iteration, we copy the value stored in the current source block to the current destination block. The addresses of both is calculated by a increment factor %esi added to the current address(%edx for src and %ebc for dest).

Finally, we resume the esp and ebp and return.

2.1.2 Code

- **sum.y**

```

1 #Execution begins at address 0
2 .pos 0 #start address for all Y86 programs
3 Init:
4     irmovl Stack, %esp #Initialize stack pointer
5     irmovl Stack, %ebp #Initialize base pointer
6     jmp Main
7     halt
8
9 .align 4
10 ele1: #Elements initialization
11     .long 0x00a
12     .long ele2
13
14     .long 0x0b0
15     .long ele3
16
17     .long 0xc00
18     .long 0
19
20 Main:
21     irmovl ele1,%esi #starting pointer
22     pushl %esi
23     call sum_list
24     halt
25
26 sum_list:
27     pushl %ebp
28     rrmovl %esp, %ebp #read the stack pointer
29     mrmovl 8(%ebp),%ebx #ebx = start pointer ele1
30     irmovl $0,%eax #sum=0
31 loop: mrmovl (%ebx),%edx #The element
32     addl %edx,%eax
33     mrmovl 4(%ebx), %esi #4(%ebx) is address of next node
34     andl %esi, %esi #if %esi=zero, jump to done
35     je done #If the pointer points to zero, return
36     rrmovl %esi,%ebx
37     jmp loop
38 done: popl %esi #restore the registers
39     popl %edx
40     popl %ebx
41     rrmovl %ebp, %esp
42     popl %ebp
43     ret
44 #stack starts here and grows to lower addresses
45 .pos 0x400
46 Stack:

```

47
48

- **rsum.ys**

```
1  #Execution begins at address 0
2  .pos 0
3  Init:
4      irmovl Stack, %esp    #Initialize stack pointer
5      irmovl Stack, %ebp
6      jmp Main
7      halt
8
9  .align 4
10 ele1:
11     .long 0x00a
12     .long ele2
13 ele2:
14     .long 0x0b0
15     .long ele3
16 ele3:
17     .long 0xc00
18     .long 0
19
20
21 Main:
22     irmovl ele1, %esi    #p_ele1
23     pushl %esi
24     xorl %eax, %eax    #set eax=0
25     call rsum_list
26     halt
27
28 rsum_list:
29     pushl %ebp
30     rrmovl %esp, %ebp    #read the stack pointer
31     pushl %ebx    #save ebx
32     pushl %ecx    #save ecx
33     pushl %edx    #save edx
34     pushl %esi    #save esi
35     mrmovl 8(%ebp), %edx    #edx=p_ele[i]
36     mrmovl 0(%edx), %eax    #eax=ele[i]
37     mrmovl 4(%edx), %ebx    #ebx=p_ele[i+1]
38     andl %ebx, %ebx    #if p_ele[i+1] == 0
39     je done    #return ele[i]
40     pushl %ebx    #else: 8(%ebp)=p_ele[i+1]
41     rrmovl %eax, %ecx    #ecx = ele[i]
42     call rsum_list
43     popl %edx    #restore the stack pointer
44     addl %ecx, %eax    #eax += rsum(p_ele[i+1])
45 done:    #return
46     popl %esi    #restore the registers
47     popl %edx
48     popl %ecx
49     popl %ebx
50     rrmovl %ebp, %esp
51     popl %ebp
52     ret
```

```

53         .pos      0x120
54 Stack:
55

```

- copy.y

```

1 #Execution begins at address 0
2     .pos      0
3 Init:
4     irmovl   Stack, %esp      #Initialize stack pointer
5     irmovl   Stack, %ebp
6     jmp      Main
7     halt
8
9     .align   4
10 src:
11     .long    0x00a
12     .long    0x0b0
13     .long    0xc00
14 dest:
15     .long    0x111
16     .long    0x222
17     .long    0x333
18
19
20 Main:
21     irmovl   src,%esi         #src
22     pushl    %esi
23     irmovl   dest,%esi        #dest
24     pushl    %esi
25     irmovl   $3,%esi          #len
26     pushl    %esi
27     call     copy_block
28     halt
29
30 copy_block:
31     pushl    %ebp
32     rrmovl   %esp, %ebp        #read the stack pointer
33     pushl    %ebx              #save ebx
34     pushl    %ecx              #save ecx
35     pushl    %edx              #save edx
36     pushl    %esi              #save esi
37     mrmovl   8(%ebp),%eax      #eax=len, len-1,...,0
38     irmovl   $0,%ebx          #tmp=0
39     irmovl   $0,%ecx          #ecx=0
40     irmovl   $0,%esi          #esi=0,4,8...
41 loop:
42     mrmovl   16(%ebp),%edx     #edx = p_src
43     addl     %esi,%edx         #edx = p_src_cur
44     mrmovl   0(%edx),%edx     #edx = src_cur
45     xorl     %edx,%ecx        #result ^= src_cur
46
47     mrmovl   12(%ebp),%ebx     #ebx = p_dest
48     addl     %esi,%ebx         #ebx = p_dest_cur
49     rmmovl   %edx,0(%ebx)     #*p_dest_cur = src_cur
50
51     irmovl   $1,%ebx          #eax-=1

```

```

52      subl    %ebx,%eax      #subl    %ebx,%eax -> eax = eax -
    ebx
53      je      done
54      irmovl  $4,%ebx        #tmp = 4
55      addl    %ebx,%esi      #esi+=tmp
56      jmp     loop
done:   rrmovl  %ecx,%eax
58      popl    %esi           #restore the registers
59      popl    %edx
60      popl    %ecx
61      popl    %ebx
62      rrmovl  %ebp, %esp
63      popl    %ebp
64      ret
65
66      .pos    0x120
67 Stack:

```

2.1.3 Evaluation

- sum.js

The screenshot displays the Y86 Processor simulator interface. The left pane shows the program code for 'sum.js', which includes instructions for initializing the stack pointer, pushing arguments, calling the 'sum_list' function, and restoring the registers. The right pane shows the processor state during execution.

Pipeline Registers

W	Stat	Instr	valE	valM	dstE	dstM
State	HLT	halt	00000000	00000000	----	----
Input	BUB	nop	00000000	00000000	----	----

Memory Stage

M	Stat	Instr	Cond	valE	valA	dstE	dstM
State	BUB	nop	N	00000000	00000000	----	----
Input	ACK	rrmovl	Y	000003F8	000003F8	%ebp	----

Execute Stage

E	Stat	Instr	valC	valA	valB	dstE	dstM	srcA	srcB
State	ACK	rrmovl	00000000	000003F8	00000000	%ebp	----	%esp	----
Input	ACK	pushl	00000000	00000000	000003FC	%esp	----	%ebx	%esp

Decode Stage

D	Stat	Instr	valC	valA	valB	dstE	dstM	srcA	srcB
State	ACK	pushl	%ebx	----	00000000	00000040	----	%esp	----
Input	ACK	pushl	%edx	----	00000000	00000042	----	%esp	----

Fetch Stage

F	Stat	predPC
State	ACK	00000040
Input	ACK	00000042

Register File

%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
cba		0	0	3fc	400	14	

Condition Codes

Stat	HLT
2	1 8 0 0 0

Performance

Cycles	Instructions	CPI
52	42	1.24

```

chacha@chacha-System-Product-Name:~/Downloads/project1-handout/pr
sim/misc$ ./yis sum.yo
Stopped in 42 steps at PC = 0x39. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%esp: 0x00000000      0x000003fc
%ebp: 0x00000000      0x00000400
%esi: 0x00000000      0x00000014

Changes to memory:
0x03e8: 0x00000000      0x00000014
0x03f4: 0x00000000      0x00000400
0x03f8: 0x00000000      0x00000039
0x03fc: 0x00000000      0x00000014

```

As is shown above, our implementation can be seen as successful.

- rsum.yis

Program Code

File	Load
0x0 30f420010000	irmovl Stack, %esp #Initialize stack pointer
0x6 30f520010000	irmovl Stack, %esp
0xc 702c000000	jmp Main
0x11 00	halt
0x14 0a000000	.long 0a00
0x18 1c000000	.long 1c00
0x1c b0000000	.long 0b00
0x20 24000000	.long 2400
0x24 00000000	.long 0a00
0x28 00000000	.long 0
0x2c 30f614000000	irmovl %eax, %esi #p_e1e1
0x32 a06f	pushl %esi
0x34 803a000000	call copy_list
0x39 00	halt
0x3a a05e	pushl %ebp
0x3c 2045	rrmovl %esp, %ebp #read the stack pointer
0x3e a03f	pushl %ebx #save ebx
0x40 a01f	pushl %ecx #save ecx
0x42 a02f	pushl %edx #save edx
0x44 a06f	pushl %esi #save esi
0x46 502000000000	rrmovl 0(%ebp), %edx #edx = %e1e1
0x48 500200000000	rrmovl 0(%edx), %eax #eax = %e1e1
0x52 502040000000	rrmovl 4(%edx), %ebx #ebx = %e1e1 + 1
0x58 6233	andi %ebx, %ebx #if p_e1e1 == 0
0x5a 736c000000	je done #return %e1e1
0x5f a03f	pushl %ebx #save %e1e1 + 1
0x61 2001	rrmovl %eax, %ecx #ecx = %e1e1
0x63 803a000000	call copy_list
0x68 b02f	popl %edx #restore the stack pointer
0x6a 6010	addl %ecx, %eax #eax += rsum(p_e1e1 + 1)
0x6c b06f	popl %esi
0x6e b02f	popl %edx
0x70 b01f	popl %ecx
0x72 b03f	popl %ebx
0x74 2004	rrmovl %ebp, %esp
0x76 b05e	popl %ebp
0x78 90	ret

Y86 Processor: pipe-full.hcl

Quit Go Stop Step Reset

Pipeline Registers

Stat	Instr	valE	valM	dstE	dstM
W	HLT	halt	00000000	00000000	----
Input	SUB	nop	00000000	00000000	----

Memory Stage

Stat	Instr	Cnd	valE	valA	dstE	dstM
M	SUB	nop	N	00000000	00000000	----
Input	AKK	rrmovl	Y	00000118	00000118	%ebp

Execute Stage

Stat	Instr	valC	valA	valB	dstE	dstM	srcA	srcB
E	AKK	rrmovl	00000000	00000118	00000000	%ebp	----	%esp
Input	AKK	pushl	00000000	00000000	0000011C	%esp	----	%ebx

Decode Stage

Stat	Instr	ra	rb	valC	valP
D	AKK	pushl	%ebx	----	00000000
Input	AKK	pushl	%ecx	----	00000000

Fetch Stage

Stat	predPC
F	AKK 00000040
Input	AKK 00000042

Register File

%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
cba	0	0	0	11c	120	14	

Stat HLT Condition Codes Z 0 S 0 O 0

Performance Cycles 90 Instructions 71 CPI 1.27

```

chacha@chacha-system-Product-Name:~/Downloads/project1-handout/project1-handout/sim/misc$ ./yas rsum.y
chacha@chacha-system-Product-Name:~/Downloads/project1-handout/project1-handout/sim/misc$ ./yis rsum.yo
Stopped in 71 steps at PC = 0x39. Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x000000c0
%esp: 0x00000000      0x0000011c
%ebp: 0x00000000      0x00000120
%esi: 0x00000000      0x00000014

Changes to memory:
0x00cc: 0x00000000      0x00000014
0x00d0: 0x00000000      0x0000001c
0x00d4: 0x00000000      0x000000b0
0x00d8: 0x00000000      0x00000024
0x00dc: 0x00000000      0x000000f8
0x00e0: 0x00000000      0x00000068
0x00e4: 0x00000000      0x00000024
0x00e8: 0x00000000      0x00000014
0x00ec: 0x00000000      0x00000014
0x00f0: 0x00000000      0x0000000a
0x00f4: 0x00000000      0x0000001c
0x00f8: 0x00000000      0x00000114
0x00fc: 0x00000000      0x00000068
0x0100: 0x00000000      0x0000001c
0x0104: 0x00000000      0x00000014
0x0114: 0x00000000      0x00000120
0x0118: 0x00000000      0x00000039
0x011c: 0x00000000      0x00000014

```

As is shown above, our implementation can be seen as successful.

- **copy.y**

```

chacha@chacha-system-Product-Name:~/Downloads/project1-handout/project1-handout/sim/misc$ ./yas copy.y
chacha@chacha-system-Product-Name:~/Downloads/project1-handout/project1-handout/sim/misc$ ./yis copy.yo
Stopped in 65 steps at PC = 0x49. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x000000c0
%esp: 0x00000000      0x00000114
%ebp: 0x00000000      0x00000120
%esi: 0x00000000      0x00000003

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x000000c0
0x00fc: 0x00000000      0x00000003
0x010c: 0x00000000      0x00000120
0x0110: 0x00000000      0x00000049
0x0114: 0x00000000      0x00000003
0x0118: 0x00000000      0x00000020
0x011c: 0x00000000      0x00000014

```

As is shown above, our implementation can be seen as successful.

Target register is returned with correct value and also, from the GUI interface, we could see the performance of CPU as well as the decoded machine language denoted by hexadecimal numbers.

2.2 Part B

An operation *iaddl* added to the control file *seq-full.hcl* to extend the SEQ processor is required in this part. (Also, leave instruction is also implemented although it is not used in our y86 programs so we are not going to details about leave.)

2.2.1 Analysis

To add *iaddl* to the SEQ processor, the steps is as follows:

1. $M_1[PC]$ is used to get the icode and ifun which combine a byte.
2. As indicated in the binary code of *iaddl*, the lower 4 bits of $M_1[pC+1]$ are used, so register rB is the register to add with the constant value, while

the higher 4 bits of $M1[pC+1]$ is F, indicating rA is not valid. Then, we get the rest lower 4 bytes of the instruction to get the instant value.

3. Decode the instruction by which we could get the value in the register and store it in the valB.
4. Execute the add operation.
5. Write the result back to the register and Finally update the PC to prepare for the next instruction. The updated PC should be PC+6 since the length of iaddl instruction is designed to be 6 bytes.

2.2.2 Code

```

1  /* $begin seq-all-hcl */
2  #####
3  #Descriptions:
4  #1.iaddl(6bytes) c0 FrB vv vv vv vv    2.leave(1byte) d0
5  # result:rB += ConstV                  result:%esp=%ebp+4;
6  #F: icode:ifun<- M1[PC]                %ebp=M[%ebp]
7  # rA:rB<-M1[PC+1]                      F: icode:ifun<-M1[PC]
8  # valC<-M4[PC+2]                        valP<-PC+1
9  # valP<-PC+6                            D: valA<-R[%esp]
10 #D: valB<-R[rB]                          E: valE<-valA+4
11 #E: valE<-valB+valC                      M: valM<-M[%ebp]
12 #M:                                       WB:R[%esp]<-valE
13 #WB:R[rB]<-valE                          R[%ebp]<-valM
14 #####
15 #iaddl Details:
16 # iaddl is similar to both addl and irmovl. Actually we can replace#
17 # this instruction with a combination of irmovl and addl. So we can#
18 # refer to IIRMOVL and IOPL to implement IIADDL. Here are the
19 # the related modifications below.
20 # 1.instr_vali
21 # 2.need_regids
22 # 3.need_valC
23 # 4.srcB - rB
24 # 5.dstE - rB
25 # 6.aluA - valC
26 # 7.aluB - valB
27 # 8.set_cc
28 #leave Details:
29 # leave is a little more complicated. But by careful analysis, we
30 # can have it implemented with some modifications below.
31 # 1.instr_vali
32 # 2.need_regids
33 # 3.srcA - REBP
34 # 4.dstE - RESP
35 # 5.dstM - REBP
36 # 6.aluA - valA
37 # 7.aluB - 4
38 # 8.mem_read
39 # 9.mem_addr - valA
40 #####

```

```

41 #####
42
43
44 #####
45 # C Include's. Don't alter these #
46 #####
47
48 quote '#include <stdio.h>'
49 quote '#include "isa.h"'
50 quote '#include "sim.h"'
51 quote 'int sim_main(int argc, char *argv[]);'
52 quote 'int gen_pc(){return 0;}'
53 quote 'int main(int argc, char *argv[])'
54 quote ' {plusmode=0;return sim_main(argc,argv);}'
55
56 #####
57 # Declarations. Do not change/remove/delete any of these #
58 #####
59
60 ##### Symbolic representation of Y86 Instruction Codes #####
61 intsig INOP 'I_NOP'
62 intsig IHALT 'I_HALT'
63 intsig IRRMOVL 'I_RRMOVL'
64 intsig IIRMOVL 'I_IRMOVL'
65 intsig IRMMOVL 'I_RMMOVL'
66 intsig IRRMOVL 'I_MRM OVL'
67 intsig IOPL 'I_ALU'
68 intsig IJXX 'I_JMP'
69 intsig ICALL 'I_CALL'
70 intsig IRET 'I_RET'
71 intsig IPUSHL 'I_PUSHL'
72 intsig IPOPL 'I_POPL'
73 # Instruction code for iaddl instruction
74 intsig IIADDL 'I_IADDL'
75 # Instruction code for iaddl instruction
76 intsig ILEAVE 'I_LEAVE'
77
78 ##### Symbolic representations of Y86 function codes
79 #####
80 intsig FNONE 'F_NONE' # Default function code
81
82 ##### Symbolic representation of Y86 Registers referenced explicitly
83 #####
84 intsig RESP 'REG_ESP' # Stack Pointer
85 intsig REBP 'REG_EBP' # Frame Pointer
86 intsig RNONE 'REG_NONE' # Special value indicating "no register"
87
88
89 ##### ALU Functions referenced explicitly
90 #####
91 intsig ALUADD 'A_ADD' # ALU should add its arguments
92
93 ##### Possible instruction status values
94 #####
95 intsig SAOK 'STAT_AOK' # Normal execution
96 intsig SADR 'STAT_ADR' # Invalid memory address
97 intsig SINS 'STAT_INS' # Invalid instruction

```

```

93 intsig SHLT 'STAT_HLT' # Halt instruction encountered
94
95 ##### Signals that can be referenced by control logic
96 #####
97 ##### Fetch stage inputs #####
98 intsig pc 'pc' # Program counter
99 ##### Fetch stage computations #####
100 intsig imem_icode 'imem_icode' # icode field from instruction
    memory
101 intsig imem_ifun 'imem_ifun' # ifun field from instruction
    memory
102 intsig icode 'icode' # Instruction control code
103 intsig ifun 'ifun' # Instruction function
104 intsig rA 'ra' # rA field from instruction
105 intsig rB 'rb' # rB field from instruction
106 intsig valC 'valc' # Constant from instruction
107 intsig valP 'valp' # Address of following instruction
108 boolsig imem_error 'imem_error' # Error signal from instruction
    memory
109 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
110
111 ##### Decode stage computations #####
112 intsig valA 'vala' # Value from register A port
113 intsig valB 'valb' # Value from register B port
114
115 ##### Execute stage computations #####
116 intsig valE 'vale' # Value computed by ALU
117 boolsig Cnd 'cond' # Branch test
118
119 ##### Memory stage computations #####
120 intsig valM 'valm' # Value read from memory
121 boolsig dmem_error 'dmem_error' # Error signal from data memory
122
123
124 #####
125 # Control Signal Definitions. #
126 #####
127
128 ##### Fetch Stage #####
129
130 # Determine instruction code
131 int icode = [
132     imem_error: INOP;
133     1: imem_icode; # Default: get from instruction memory
134 ];
135
136 # Determine instruction function
137 int ifun = [
138     imem_error: FNONE;
139     1: imem_ifun; # Default: get from instruction memory
140 ];
141
142 bool instr_valid = icode in
143     { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL, IIADDL, ILEAVE,
144       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
145

```

```

146 # Does fetched instruction require a regid byte?
147 bool need_regids =
148     icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIADDL, ILEAVE,
149             IIRMOVL, IRMMOVL, IMRMOVL };
150
151 # Does fetched instruction require a constant word?
152 bool need_valC =
153     icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
154
155 ##### Decode Stage #####
156
157 ## What register should be used as the A source?
158 int srcA = [
159     icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
160     icode in { IPOPL, IRET } : RESP;
161     icode in { ILEAVE } : REBP;
162     1 : RNONE; # Don't need register
163 ];
164
165 ## What register should be used as the B source?
166 int srcB = [
167     icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
168     icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
169     1 : RNONE; # Don't need register
170 ];
171
172 ## What register should be used as the E destination?
173 int dstE = [
174     icode in { IRRMOVL } && Cnd : rB;
175     icode in { IIRMOVL, IOPL, IIADDL } : rB;
176     icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
177     1 : RNONE; # Don't write any register
178 ];
179
180 ## What register should be used as the M destination?
181 ## Acquire data from memory, and send it to a register
182 ## iaddl doesn't require memory's data
183 int dstM = [
184     icode in { IMRMOVL, IPOPL } : rA;
185     icode in { ILEAVE } : REBP;
186     1 : RNONE; # Don't write any register
187 ];
188
189 ##### Execute Stage #####
190
191 ## Select input A to ALU
192 int aluA = [
193     icode in { IRRMOVL, IOPL } : valA;
194     icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
195     icode in { ICALL, IPUSHL } : -4;
196     icode in { IRET, IPOPL, ILEAVE } : 4;
197     # Other instructions don't need ALU
198 ];
199
200 ## Select input B to ALU
201 int aluB = [
202     icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,

```

```

203         IPUSHL, IRET, IPOPL, IIADDL } : valB;
204     icode in { IRRMOVL, IIRMOVL } : 0;
205     icode in { ILEAVE } : valA;
206     # Other instructions don't need ALU
207 ];
208
209 ## Set the ALU function
210 int alufun = [
211     icode == IOPL : ifun;
212     1 : ALUADD;
213 ];
214
215 ## Should the condition codes be updated?
216 bool set_cc = icode in { IOPL, IIADDL };
217
218 ##### Memory Stage #####
219
220 ## Set read control signal
221 bool mem_read = icode in { IRRMOVL, IPOPL, IRET, ILEAVE };
222
223 ## Set write control signal
224 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
225
226 ## Select memory address
227 int mem_addr = [
228     icode in { IRMMOVL, IPUSHL, ICALL, IRRMOVL } : valE;
229     icode in { IPOPL, IRET, ILEAVE } : valA;
230     # Other instructions don't need address
231 ];
232
233 ## Select memory input data
234 int mem_data = [
235     # Value from register
236     icode in { IRMMOVL, IPUSHL } : valA;
237     # Return PC
238     icode == ICALL : valP;
239     # Default: Don't write anything
240 ];
241
242 ## Determine instruction status
243 int Stat = [
244     imem_error || dmem_error : SADR;
245     !instr_valid : SINS;
246     icode == IHALT : SHLT;
247     1 : SAOK;
248 ];
249
250 ##### Program Counter Update #####
251
252 ## What address should instruction be fetched at
253
254 int new_pc = [
255     # Call. Use instruction constant
256     icode == ICALL : valC;
257     # Taken branch. Use instruction constant
258     icode == IJXX && Cnd : valC;
259     # Completion of RET instruction. Use value from stack

```

```

260 icode == IRET : valM;
261  # Default: Use incremented PC
262  1 : valP;
263  ];
264  /* Send seq-all-hcl */

```

2.2.3 Evaluation

- Test for *asumi* & *asuml*

The image displays two screenshots of the Y86 Processor simulator, showing the execution of assembly code. The top screenshot shows the initial state with PC at 0000003A and registers rax, rcx, rdx, rdi, rsi, rbp, rdi. The bottom screenshot shows the processor at a later stage with PC at 00000096 and registers rax, rcx, rdx, rdi, rsi, rbp, rdi. Both screenshots show the Program Code window with assembly instructions and the Processor State window with various stages like PC Update, Memory, Execute, Decode, and Fetch.

Top Screenshot:

- Program Code:**

```

0x0 30f400010000  init: irmovl Stack, %esp    # Set up Stack pointer
0x1 30f500010000  irmovl Stack, %ebp    # Set up base pointer
0x2 702400000000  jmp Main              # Execute main program
0x3 0a0000000000  array: .long 0xd
0x4 c00000000000  .long 0xc0
0x5 000b00000000  .long 0xb00
0x6 00a000000000  .long 0xa000
0x7 30f004000000  Main: irmovl %4,%eax
0x8 a00f          pushl %eax          # Push 4
0x9 30f214000000  irmovl array,%edx
0xa a02f          pushl %edx          # Push array
0xb 803a00000000  call %eax          # Sum(array, 4)
0xc 00          halt
0xd a05f          Sum: pushl %ebp
0xe 2045          rrmovl %esp,%ebp
0xf 501508000000  rmmovl 8(%ebp),%ecx  # ecx = Start
0x10 50250c000000  rmmovl 12(%ebp),%edx # edx = Count
0x11 30f000000000  irmovl %0,%eax      # sum = 0
0x12 6222          andl %edx,%edx
0x13 737000000000  je %end
0x14 306100000000  Loop: mrmovl (%eax),%esi # get *Start
0x15 6060          addl %esi,%eax      # add to sum
0x16 c0f104000000  iaddl %4,%ecx        # Sum++
0x17 c0f2ffffff00  iaddl %-1,%edx       # Count--
0x18 745700000000  jne Loop            # Stop when 0
0x19 b05f          End: popl %ebp
0x20 90          ret

```
- Processor State:**
 - newPC: 0000003A
 - PC Update Stage: valM: 00000000
 - Memory Stage: Cnd: valE: N 00000000
 - Execute Stage: valA: 00000000, valB: 00000000, valC: 00000000, valD: 00000000
 - Decode Stage: Instr: halt, rA: 0, rB: 0, valC: 00000000, valP: 0000003A
 - Fetch Stage: PC: 00000039
- Register File:**

%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
abcd	24	0		38	100	a000	
- Condition Codes:** Z=0, O=0, B=0, S=0, P=0

Bottom Screenshot:

- Program Code:**

```

0x0 30f400010000  init: irmovl Stack, %esp    # Set up Stack pointer
0x1 30f500010000  irmovl Stack, %ebp    # Set up base pointer
0x2 702400000000  jmp Main              # Execute main program
0x3 0a0000000000  array: .long 0xd
0x4 c00000000000  .long 0xc0
0x5 000b00000000  .long 0xb00
0x6 00a000000000  .long 0xa000
0x7 30f004000000  Main: irmovl %4,%eax
0x8 a00f          pushl %eax          # Push 4
0x9 30f214000000  irmovl array,%edx
0xa a02f          pushl %edx          # Push array
0xb 803a00000000  call %eax          # Sum(array, 4)
0xc 00          halt
0xd a05f          rsum: pushl %ebp
0xe 2045          rrmovl %esp,%ebp
0xf 30f014000000  irmovl %20,%eax
0x10 6104          subl %eax,%esp
0x11 a03f          pushl %eax
0x12 503508000000  rmmovl 8(%ebp),%ecx  # add to sum
0x13 50050c000000  rmmovl 12(%ebp),%edx # Sum++
0x14 6200          andl %eax,%eax
0x15 71b000000000  jle L38
0x16 30f200000000  irmovl %-8,%edx
0x17 6024          addl %edx,%esp
0x18 30f2ffffff00  irmovl %-1,%edx
0x19 6020          addl %edx,%eax
0x1a a00f          pushl %eax
0x1b 30f204000000  irmovl %4,%edx
0x1c 2030          rrmovl %ebx,%eax
0x1d 6020          addl %edx,%eax
0x1e a00f          pushl %eax
0x1f 803a00000000  call %eax
0x20 502300000000  rmmovl (%eax),%edx
0x21 6020          addl %edx,%eax
0x22 708400000000  jmp L39
0x23 6300          L38: xchgl %eax,%eax
0x24 50350c000000  L39: rmmovl -24(%ebp),%ecx
0x25 d0          leave
0x26 90          ret

```
- Processor State:**
 - newPC: 00000096
 - PC Update Stage: valM: 00000000
 - Memory Stage: Cnd: valE: N 00000000
 - Execute Stage: valA: 00000000, valB: 00000000, valC: 00000000, valD: 00000000
 - Decode Stage: Instr: halt, rA: 0, rB: 0, valC: 00000000, valP: 00000096
 - Fetch Stage: PC: 00000095
- Register File:**

%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
0	4	20	334	360			
- Condition Codes:** Z=1, O=0, B=0, S=0, P=0

- Retest using the benchmark programs

```

File Edit View Search Terminal Help
lingzx@ubuntu: ~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq
Couldn't open object file asunr.yo
lingzx@ubuntu:~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq$ ./ssn -g /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/y86-code/asunr.yo
lingzx@ubuntu:~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq$ (cd /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/y86-code; make testssn)
../seq/ssn -t asunr.yo > asunr.seq
../seq/ssn -t cjr.yo > cjr.seq
../seq/ssn -t j-cc.yo > j-cc.seq
../seq/ssn -t poptest.yo > poptest.seq
../seq/ssn -t pushquestion.yo > pushquestion.seq
../seq/ssn -t prog1.yo > prog1.seq
../seq/ssn -t prog2.yo > prog2.seq
../seq/ssn -t prog3.yo > prog3.seq
../seq/ssn -t prog4.yo > prog4.seq
../seq/ssn -t prog5.yo > prog5.seq
../seq/ssn -t prog6.yo > prog6.seq
../seq/ssn -t prog7.yo > prog7.seq
../seq/ssn -t prog8.yo > prog8.seq
../seq/ssn -t ret-hazard.yo > ret-hazard.seq
grep "ISA Check" *.seq
asunr.seq:ISA Check Succeeds
asunr.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
pushquestion.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asunr.seq asunr.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushquestion.seq prog1.seq prog2.seq prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq prog8.seq ret-hazard.seq
lingzx@ubuntu:~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq$

```

- Perform regression tests

```

File Edit View Search Terminal Help
lingzx@ubuntu: ~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq
../seq/ssn -t prog3.yo > prog3.seq
../seq/ssn -t prog4.yo > prog4.seq
../seq/ssn -t prog5.yo > prog5.seq
../seq/ssn -t prog6.yo > prog6.seq
../seq/ssn -t prog7.yo > prog7.seq
../seq/ssn -t prog8.yo > prog8.seq
../seq/ssn -t ret-hazard.yo > ret-hazard.seq
grep "ISA Check" *.seq
asunr.seq:ISA Check Succeeds
asunr.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
pushquestion.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asunr.seq asunr.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushquestion.seq prog1.seq prog2.seq prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq prog8.seq ret-hazard.seq
lingzx@ubuntu:~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq$ (cd /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/test; make SIM=/home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn; TFLAGS=-ll)
/optest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
All 49 ISA Checks Succeed
./jtest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
All 64 ISA Checks Succeed
./ctest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
All 22 ISA Checks Succeed
./htest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq/ssn
All 600 ISA Checks Succeed
lingzx@ubuntu:~/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/seq$

```

2.3 Part C

2.3.1 Analysis

In this part, we are required to optimize the performance of a function `ncopy`, which copies the data from source address to destine address and return the number of positive integers contained in the source. And to achieve this goal,

optimization of both algorithm and hardware is allowed. So, this part is a test of our overall capability of pipeline architecture.

The performance of the function is evaluated with CPE, so what we need to do is to reduce average CPE as more as possible. The difficulties lie in several aspects below, and we have also figured out the answer.

1. What makes the function perform poorly? A: Great number of branch instructions, high cost of computation involving immediate integers and stall penalty from load-read instructions. We are not talking about misprediction penalty of conditional branches but just the proportion of branch instructions that would cost a lot of cycles.
2. How can we reduce branch instructions, computation cost and stall penalty? A: Reduce instructions of conditional branches to improve our algorithm, add new instruction(s) to increase support for immediate computation and adjust sequence of some instructions.
3. What should we do in software layer? A: Modify `ncopy.js`, and apply technique of loop unrolling to reduce number of branch instructions, use instruction(s) that supports immediate computation better when necessary and adjust sequence of some instructions that would cause a data hazard.
4. What should we do in hardware layer? A: Modify `pipe-full.hcl`, and implement logic that supports immediate computation, that is `iaddl`. (Apart from `iaddl`, `ileave` is also implemented here according to the requirement but it is found to be of no use in this part.) Now we will elaborate what we do here.

Firstly, loop unrolling. Technique of loop unrolling reduces the number of branch instructions and thus reduce the number of instructions to execute. We have a loop that performs `ncopy` of 16 elements. In the primitive version of `ncopy` every time a number is copied there would be a check whether the loop should be over. Thus, we reduce the number of instructions by 15 every 16 elements, which also means that the CPE could be decreased by about 15/16 with technique of loop unrolling. Also, we need to tie up some loose ends. To achieve better performance, after the 16-element loop, we do the `ncopy` work with 8, 4, 2 and 1 element(s) successively if there are that many elements left.

Secondly, use `iaddl` for immediate computation. Decreasing of `len` and increasing of `count`, `p_src` and `p_dst` are involved with immediate operands. We could have CPE decreased by 2 with this step.

Thirdly, avoid load-read stall penalty. It is easy to find a “`mrmovl x1, x2`” instruction followed by a “`rmmovl x2, x3`” instruction, which intends to copy `*p_x1` to `*p_x3`. But since `mrmovl` is a load instruction and `rmmovl` needs to read the same register. So, codes like this would cause a penalty of one cycle. In other words, by inserting some other instructions into the two instructions can decrease the CPE approximately by 1.

Fourthly, implementation of `iaddl` and `ileave`. Detailed descriptions of `iaddl` and `leave` can be seen in the beginning part of `seq-full.hcl` and `pipe-full.hcl`. (Although we are talking about implementation of pipeline processor, but the operations of the two instructions are similar to that of a sequence processor). `iaddl`, which adds an immediate operand to a register, can be accomplished by combination of `irmovl` and `addl`. `leave`, which decrease stack pointer and load data to base pointer register memory addressed by itself, can be accomplished by combination of `mrmovl` and `popl`. Inspired by instructions similar to them, we could modify the `hcl` file properly. Here are some further details. `iaddl`: `instr_vali`, `need_regids`, `need_valC`, `d_srcB` = `D_rB`, `d_dstE` = `D_rB`, `aluA` = `valC`, `aluB` = `valB`, `set_cc`. `leave`: `instr_vali`, `need_regids`, `d_srcA` = `REBP`, `d_dstE` = `RESP`, `d_dstM` = `REBP`, `aluA` = `E_valA`, `aluB` = 4, `mem_addr` = `M_valA`, `mem_read`, `F_stall`, `D_stall`, `D_bubble`, `E_bubble`. Using `iaddl` instruction can also reduce CPE by about 2.

2.3.2 Code

- `ncopy.y`

```

1  /* $begin ncopy-y */
2  # 948bytes < 1000bytes
3  # Average CPE = 9.89
4  # The trick used in the ncopy function is loop unrolling.
5  # 1. First we iteratively 'ncopy' 16 elements until there are fewer
6  # than 16 elements left.
7  # 2. Then check if left elements are more than 8, and if so,
8  # we 'ncopy' 8 elements.
9  # 3. Repeat procedure2 by checking and 'ncopy' 4, 2, 1 element(s).
10 # The modifications above reduces number of condition branch
11 # instructions, and thus improve the performance of CPU.
12 # Also, some sequences of instructions are modified.
13 # Src-plus and count-plus instructions are inserted between
14 # some mrmovl and rmmovl instructions to avoid a stalling
15 # due to data hazard.
16 #####
17 # ncopy.y - Copy a src block of len ints to dst.
18 # Return the number of positive ints (>0) contained in src.
19 #
20 # Include your name and ID here.
21 #
22 # Describe how and why you modified the baseline code.
23 #
24 #####
25 # Do not modify this portion
26 # Function prologue.
27 ncopy:  pushl %ebp      # Save old frame pointer
28         rmmovl %esp,%ebp  # Set up new frame pointer
29         pushl %esi      # Save callee-save regs
30         pushl %ebx
31         pushl %edi
32         mrmovl 8(%ebp),%ebx # src
33         mrmovl 16(%ebp),%edx # len

```

```

34      mrmovl 12(%ebp),%ecx    # dst
35
36 #####
37 # You can modify this portion
38 xorl %eax,%eax             # count = 0;
39
40 ##### Iteratively 'ncopy' 16 elements until #####
41 ##### fewer than 16 elements left. #####
42 Loop5: iaddl $-16,%edx      # len-=16 > 0 ?
43      jle Loop4              # if so, goto Loop:
44
45      mrmovl (%ebx), %esi    # read val from src...
46      andl %esi, %esi        # val <= 0?
47      jle Npos51             # if so, goto Npos:
48      iaddl $1, %eax          # count++
49 Npos51: rmmovl %esi, (%ecx)  # ...and store it to dst
50
51      mrmovl 4(%ebx), %esi    # read val from src...
52      andl %esi, %esi        # val <= 0?
53      jle Npos52             # if so, goto Npos:
54      iaddl $1, %eax          # count++
55 Npos52: rmmovl %esi, 4(%ecx) # ...and store it to dst
56
57      mrmovl 8(%ebx), %esi    # read val from src...
58      andl %esi, %esi        # val <= 0?
59      jle Npos53             # if so, goto Npos:
60      iaddl $1, %eax          # count++
61 Npos53: rmmovl %esi, 8(%ecx) # ...and store it to dst
62
63      mrmovl 12(%ebx), %esi   # read val from src...
64      andl %esi, %esi        # val <= 0?
65      jle Npos54             # if so, goto Npos:
66      iaddl $1, %eax          # count++
67 Npos54: rmmovl %esi, 12(%ecx) # ...and store it to dst
68
69      mrmovl 16(%ebx), %esi   # read val from src...
70      andl %esi, %esi        # val <= 0?
71      jle Npos55             # if so, goto Npos:
72      iaddl $1, %eax          # count++
73 Npos55: rmmovl %esi, 16(%ecx) # ...and store it to dst
74
75      mrmovl 20(%ebx), %esi   # read val from src...
76      andl %esi, %esi        # val <= 0?
77      jle Npos56             # if so, goto Npos:
78      iaddl $1, %eax          # count++
79 Npos56: rmmovl %esi, 20(%ecx) # ...and store it to dst
80
81      mrmovl 24(%ebx), %esi   # read val from src...
82      andl %esi, %esi        # val <= 0?
83      jle Npos57             # if so, goto Npos:
84      iaddl $1, %eax          # count++
85 Npos57: rmmovl %esi, 24(%ecx) # ...and store it to dst
86
87      mrmovl 28(%ebx), %esi   # read val from src...
88      andl %esi, %esi        # val <= 0?
89      jle Npos58             # if so, goto Npos:
90      iaddl $1, %eax          # count++

```

```

91 Npos58: rmmovl %esi, 28(%ecx)  # ...and store it to dst
92
93     mrmovl 32(%ebx), %esi  # read val from src...
94     andl %esi, %esi      # val <= 0?
95     jle Npos59           # if so, goto Npos:
96     iaddl $1, %eax       # count++
97 Npos59: rmmovl %esi, 32(%ecx)  # ...and store it to dst
98
99     mrmovl 36(%ebx), %esi  # read val from src...
100    andl %esi, %esi      # val <= 0?
101    jle Npos510          # if so, goto Npos:
102    iaddl $1, %eax       # count++
103 Npos510: rmmovl %esi, 36(%ecx)  # ...and store it to dst
104
105    mrmovl 40(%ebx), %esi  # read val from src...
106    andl %esi, %esi      # val <= 0?
107    jle Npos511          # if so, goto Npos:
108    iaddl $1, %eax       # count++
109 Npos511: rmmovl %esi, 40(%ecx)  # ...and store it to dst
110
111    mrmovl 44(%ebx), %esi  # read val from src...
112    andl %esi, %esi      # val <= 0?
113    jle Npos512          # if so, goto Npos:
114    iaddl $1, %eax       # count++
115 Npos512: rmmovl %esi, 44(%ecx)  # ...and store it to dst
116
117    mrmovl 48(%ebx), %esi  # read val from src...
118    andl %esi, %esi      # val <= 0?
119    jle Npos513          # if so, goto Npos:
120    iaddl $1, %eax       # count++
121 Npos513: rmmovl %esi, 48(%ecx)  # ...and store it to dst
122
123    mrmovl 52(%ebx), %esi  # read val from src...
124    andl %esi, %esi      # val <= 0?
125    jle Npos514          # if so, goto Npos:
126    iaddl $1, %eax       # count++
127 Npos514: rmmovl %esi, 52(%ecx)  # ...and store it to dst
128
129    mrmovl 56(%ebx), %esi  # read val from src...
130    andl %esi, %esi      # val <= 0?
131    jle Npos515          # if so, goto Npos:
132    iaddl $1, %eax       # count++
133 Npos515: rmmovl %esi, 56(%ecx)  # ...and store it to dst
134
135    mrmovl 60(%ebx), %esi  # read val from src...
136    iaddl $64, %ebx       # src+=16
137    rmmovl %esi, 60(%ecx)  # ...and store it to dst
138    andl %esi, %esi      # val <= 0?
139    jle Npos516          # if so, goto Npos:
140    iaddl $1, %eax       # count++
141 Npos516: iaddl $64, %ecx    # dst+=16
142    jmp Loop5            # goto Loop:
143 ##### Iterative 'ncopy' of 16 elements is over #####
144
145
146
147 ### Check if left elements are more than 8, and if so, #####

```

```

148 ### we 'ncopy' 8 elements. #####
149 Loop4: iaddl $8,%edx # len-=4 > 0 ?
150     jl Loop3 # if so, goto Loop:
151
152     mrmovl (%ebx), %esi # read val from src...
153     andl %esi, %esi # val <= 0?
154     jle Npos41 # if so, goto Npos:
155     iaddl $1, %eax # count++
156 Npos41: rmmovl %esi, (%ecx) # ...and store it to dst
157
158     mrmovl 4(%ebx), %esi # read val from src...
159     andl %esi, %esi # val <= 0?
160     jle Npos42 # if so, goto Npos:
161     iaddl $1, %eax # count++
162 Npos42: rmmovl %esi, 4(%ecx) # ...and store it to dst
163
164     mrmovl 8(%ebx), %esi # read val from src...
165     andl %esi, %esi # val <= 0?
166     jle Npos43 # if so, goto Npos:
167     iaddl $1, %eax # count++
168 Npos43: rmmovl %esi, 8(%ecx) # ...and store it to dst
169
170     mrmovl 12(%ebx), %esi # read val from src...
171     andl %esi, %esi # val <= 0?
172     jle Npos44 # if so, goto Npos:
173     iaddl $1, %eax # count++
174 Npos44: rmmovl %esi, 12(%ecx) # ...and store it to dst
175
176     mrmovl 16(%ebx), %esi # read val from src...
177     andl %esi, %esi # val <= 0?
178     jle Npos45 # if so, goto Npos:
179     iaddl $1, %eax # count++
180 Npos45: rmmovl %esi, 16(%ecx) # ...and store it to dst
181
182     mrmovl 20(%ebx), %esi # read val from src...
183     andl %esi, %esi # val <= 0?
184     jle Npos46 # if so, goto Npos:
185     iaddl $1, %eax # count++
186 Npos46: rmmovl %esi, 20(%ecx) # ...and store it to dst
187
188     mrmovl 24(%ebx), %esi # read val from src...
189     andl %esi, %esi # val <= 0?
190     jle Npos47 # if so, goto Npos:
191     iaddl $1, %eax # count++
192 Npos47: rmmovl %esi, 24(%ecx) # ...and store it to dst
193
194     mrmovl 28(%ebx), %esi # read val from src...
195     iaddl $32,%ebx # src+=8
196     rmmovl %esi, 28(%ecx) # ...and store it to dst
197     andl %esi, %esi # val <= 0?
198     jle Npos48 # if so, goto Npos:
199     iaddl $1, %eax # count++
200 Npos48: iaddl $32,%ecx # dst+=8
201     iaddl $-8,%edx
202 ##### End of 8-element checking and 'ncopy #####
203
204

```

```

205 ##### Check if left elements are more than 4, and if so, #####
206 ##### we 'ncopy' 4 elements. #####
207 Loop3: iaddl $4,%edx # len-=4 > 0 ?
208     jl Loop2 # if so, goto Loop:
209
210     mrmovl (%ebx), %esi # read val from src...
211     andl %esi, %esi # val <= 0?
212     jle Npos31 # if so, goto Npos:
213     iaddl $1, %eax # count++
214 Npos31: rmmovl %esi, (%ecx) # ...and store it to dst
215
216     mrmovl 4(%ebx), %esi # read val from src...
217     andl %esi, %esi # val <= 0?
218     jle Npos32 # if so, goto Npos:
219     iaddl $1, %eax # count++
220 Npos32: rmmovl %esi, 4(%ecx) # ...and store it to dst
221
222     mrmovl 8(%ebx), %esi # read val from src...
223     andl %esi, %esi # val <= 0?
224     jle Npos33 # if so, goto Npos:
225     iaddl $1, %eax # count++
226 Npos33: rmmovl %esi, 8(%ecx) # ...and store it to dst
227
228     mrmovl 12(%ebx), %esi # read val from src...
229     iaddl $16,%ebx # src+++++++
230     rmmovl %esi, 12(%ecx) # ...and store it to dst
231     andl %esi, %esi # val <= 0?
232     jle Npos34 # if so, goto Npos:
233     iaddl $1, %eax # count++
234 Npos34: iaddl $16,%ecx # dst+++++++
235     iaddl $-4,%edx
236 ##### End of 4-element checking and 'ncopy' #####
237
238
239 ##### Check if left elements are more than 2, and if so, #####
240 ##### we 'ncopy' 2 elements. #####
241 Loop2: iaddl $2,%edx # len+2 < 0 ?
242     jl Loop1 # if so, goto Loop1:
243     mrmovl (%ebx), %esi # read val from src...
244     andl %esi, %esi # val <= 0?
245     jle Npos21 # if so, goto Npos:
246     iaddl $1, %eax # count++
247 Npos21: rmmovl %esi, (%ecx) # ...and store it to dst
248     mrmovl 4(%ebx), %esi # read val from src...
249     iaddl $8,%ebx # src++++
250     rmmovl %esi, 4(%ecx) # ...and store it to dst
251     andl %esi, %esi # val <= 0?
252     jle Npos22 # if so, goto Npos:
253     iaddl $1, %eax # count++
254 Npos22: iaddl $8, %ecx # dst++++
255     iaddl $-2,%edx
256 ##### End of 2-element checking and 'ncopy' #####
257
258
259 ##### Check if left elements are more than 1, and if so, #####
260 ##### we 'ncopy' 1 elements. #####
261 Loop1: iaddl $1,%edx # len+1 < 0?

```

```

262     jl Done          # if so, goto Done:
263     rmmovl (%ebx), %esi # read val from src...
264     iaddl $4,%ebx      # src++
265     rmmovl %esi, (%ecx) # ...and store it to dst
266     andl %esi, %esi    # val <= 0?
267     jle Done          # if so, goto Npos:
268     iaddl $1, %eax     # count++
269 ##### End of 1-element checking and 'ncopy' #####
270
271 # Do not modify the following section of code
272 # Function epilogue.
273 Done:
274     popl %edi          # Restore callee-save registers
275     popl %ebx
276     popl %esi
277     rmmovl %ebp, %esp
278     popl %ebp
279     ret
280 #####
281 # Keep the following label at the end of your function
282 End:
283 /* Send ncopy-ys */
284
285

```

- **pipe-full.hcl**

```

1  /* $begin pipe-all-hcl */
2  #####
3  #   HCL Description of Control for Pipelined Y86 Processor
4  #   Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010
5  #   #####
6
7  ## Your task is to implement the iaddl and leave instructions
8  ## The file contains a declaration of the icodes
9  ## for iaddl (IIADDL) and leave (ILEAVE).
10 ## Your job is to add the rest of the logic to make it work
11
12 #####
13 #####
14 #   Leader's name and ID.
15 #   #####
16 #Descriptions:                                     #
17 # Both instructions leave and iaddl are implemented here, which
18 #   are#
19 # similar to those of 'seq'.                         #
20 # For iaddl/IIADDL, the work is almost the same as that of 'seq'
21 #   #
22 # version. The difference is that information, such as source
23 #   #
24 # register or destine register is acquired from pipeline registers
25 #   #
26 # (It's very lucky to see all forwarding logic has already been
27 #   #

```

```

23 # implmeneted)                                     #
24 # For leave/ILEAVE, the work is more complicated. Since this is a
    #
25 # load instruction, attention must be paid to avoidance of data
    #
26 # hazards. By careful analysis, we decide that ILEAVE can be
    #
27 # grouped with IMRMOVL, IPOPL when coping with data hazards, which
    #
28 # largely reduced complexity of the job.                                     #
29 #####
30 #iaddl Details:                                     #
31 # 1.instr_valid                                     #
32 # 2.need_regids                                     #
33 # 3.need_valC                                       #
34 # 4.d_srcB - D_rB                                   #
35 # 5.d_dstE - D_rB                                   #
36 # 6.aluA - valC                                     #
37 # 7.aluB - valB                                     #
38 # 8.set_cc                                           #
39 #leave Details:                                     #
40 # 1.instr_vali                                       #
41 # 2.need_regids                                     #
42 # 3.d_srcA - REBP                                   #
43 # 4.d_dstE - RESP                                   #
44 # 5.d_dstM - REBP                                   #
45 # 6.aluA - E_valA                                   #
46 # 7.aluB - 4                                         #
47 # 8.mem_addr - M_valA                               #
48 # 9.mem_read                                         #
49 # 10.F_stall                                         #
50 # 11.D_stall                                         #
51 # 12.D_bubble                                        #
52 # 13.E_bubble                                        #
53 #####
54 #####
55 #####
56 #####
57 # C Include's. Don't alter these
    #
58 #####
59
60 quote '#include <stdio.h>'
61 quote '#include "isa.h"'
62 quote '#include "pipeline.h"'
63 quote '#include "stages.h"'
64 quote '#include "sim.h"'
65 quote 'int sim_main(int argc, char *argv[]);'
66 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv)
    ;}'
67
68 #####
69 # Declarations. Do not change/remove/delete any of these
    #
70 #####
71
72 ##### Symbolic representation of Y86 Instruction Codes

```

```

#####
73 intsig INOP      'I_NOP'
74 intsig IHALT    'I_HALT'
75 intsig IRRMOVL  'I_RRMOVL'
76 intsig IIRMOVL  'I_IRMOVL'
77 intsig IRMMOVL  'I_RMMOVL'
78 intsig IMRMOVL  'I_MRMOVL'
79 intsig IOPL      'I_ALU'
80 intsig IJXX      'I_JMP'
81 intsig ICALL      'I_CALL'
82 intsig IRET      'I_RET'
83 intsig IPUSHL     'I_PUSHL'
84 intsig IPOPL      'I_POPL'
85 # Instruction code for iaddl instruction
86 intsig IIADDL     'I_IADDL'
87 # Instruction code for leave instruction
88 intsig ILEAVE     'I_LEAVE'
89
90 ##### Symbolic representations of Y86 function codes
91 #####
92 intsig FNONE      'F_NONE'          # Default function code
93 ##### Symbolic representation of Y86 Registers referenced
94 #####
95 intsig RESP       'REG_ESP'          # Stack Pointer
96 intsig REBP       'REG_EBP'          # Frame Pointer
97 intsig RNONE      'REG_NONE'         # Special value indicating "
98                                     no register"
99 ##### ALU Functions referenced explicitly
100 #####
101 intsig ALUADD      'A_ADD'            # ALU should add its arguments
102
103 ##### Possible instruction status values
104 #####
105 intsig SBUB        'STAT_BUB'        # Bubble in stage
106 intsig SAOK        'STAT_AOK'        # Normal execution
107 intsig SADR        'STAT_ADR'        # Invalid memory address
108 intsig SINS        'STAT_INS'        # Invalid instruction
109 intsig SHLT        'STAT_HLT'        # Halt instruction encountered
110
111 ##### Signals that can be referenced by control logic
112 #####
113
114 ##### Pipeline Register F
115 #####
116 intsig F_predPC    'pc_curr->pc'     # Predicted value of PC
117
118 ##### Intermediate Values in Fetch Stage
119 #####
120 intsig imem_icode   'imem_icode'      # icode field from
121                                     instruction memory
122 intsig imem_ifun    'imem_ifun'       # ifun field from
123                                     instruction memory
124 intsig f_icode      'if_id_next->icode' # (Possibly modified)

```



```

119      instruction code
120      intsig f_ifun  'if_id_next->ifun'  # Fetched instruction
      function
121      intsig f_valC  'if_id_next->valc'  # Constant data of fetched
      instruction
122      intsig f_valP  'if_id_next->valp'  # Address of following
      instruction
123      boolsig imem_error 'imem_error'    # Error signal from
      instruction memory
124      boolsig instr_valid 'instr_valid'   # Is fetched instruction
      valid?
125      ##### Pipeline Register D
      #####
126      intsig D_icode 'if_id_curr->icode'  # Instruction code
127      intsig D_rA  'if_id_curr->ra'      # rA field from instruction
128      intsig D_rB  'if_id_curr->rb'      # rB field from instruction
129      intsig D_valP 'if_id_curr->valp'    # Incremented PC
130
131      ##### Intermediate Values in Decode Stage
      #####
132
133      intsig d_srcA  'id_ex_next->srca'   # srcA from decoded
      instruction
134      intsig d_srcB  'id_ex_next->srcb'   # srcB from decoded
      instruction
135      intsig d_rvalA 'd_regvala'         # valA read from register file
136      intsig d_rvalB 'd_regvalb'         # valB read from register file
137
138      ##### Pipeline Register E
      #####
139      intsig E_icode 'id_ex_curr->icode'  # Instruction code
140      intsig E_ifun  'id_ex_curr->ifun'   # Instruction function
141      intsig E_valC  'id_ex_curr->valc'   # Constant data
142      intsig E_srcA  'id_ex_curr->srca'   # Source A register ID
143      intsig E_valA  'id_ex_curr->vala'   # Source A value
144      intsig E_srcB  'id_ex_curr->srcb'   # Source B register ID
145      intsig E_valB  'id_ex_curr->valb'   # Source B value
146      intsig E_dstE  'id_ex_curr->deste'  # Destination E register ID
147      intsig E_dstM  'id_ex_curr->destm'  # Destination M register ID
148
149      ##### Intermediate Values in Execute Stage
      #####
150      intsig e_vale  'ex_mem_next->vale'  # vale generated by ALU
151      boolsig e_Cnd  'ex_mem_next->takebranch' # Does condition hold?
152      intsig e_dstE  'ex_mem_next->deste'  # dstE (possibly modified
      to be RNONE)
153
154      ##### Pipeline Register M
      #####
155      intsig M_stat  'ex_mem_curr->status' # Instruction status
156      intsig M_icode 'ex_mem_curr->icode'  # Instruction code
157      intsig M_ifun  'ex_mem_curr->ifun'   # Instruction function
158      intsig M_valA  'ex_mem_curr->vala'   # Source A value
159      intsig M_dstE  'ex_mem_curr->deste'  # Destination E register ID
160      intsig M_valE  'ex_mem_curr->vale'   # ALU E value
161      intsig M_dstM  'ex_mem_curr->destm'  # Destination M register ID

```

```

162 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
163 boolsig dmem_error 'dmem_error' # Error signal from
    instruction memory
164
165 ##### Intermediate Values in Memory Stage
    #####
166 intsig m_valM 'mem_wb_next->valm' # valM generated by memory
167 intsig m_stat 'mem_wb_next->status' # stat (possibly modified to
    be SADR)
168
169 ##### Pipeline Register W
    #####
170 intsig W_stat 'mem_wb_curr->status' # Instruction status
171 intsig W_icode 'mem_wb_curr->icode' # Instruction code
172 intsig W_dstE 'mem_wb_curr->deste' # Destination E register ID
173 intsig W_valE 'mem_wb_curr->vale' # ALU E value
174 intsig W_dstM 'mem_wb_curr->destm' # Destination M register ID
175 intsig W_valM 'mem_wb_curr->valm' # Memory M value
176
177 #####
178 # Control Signal Definitions.
    #
179 #####
180
181 ##### Fetch Stage
    #####
182
183 ## What address should instruction be fetched at
184 int f_pc = [
185     # Mispredicted branch. Fetch at incremented PC
186     M_icode == IJXX && !M_Cnd : M_valA;
187     # Completion of RET instruction.
188     W_icode == IRET : W_valM;
189     # Default: Use predicted value of PC
190     1 : F_predPC;
191 ];
192
193 ## Determine icode of fetched instruction
194 int f_icode = [
195     imem_error : INOP;
196     1: imem_icode;
197 ];
198
199 # Determine ifun
200 int f_ifun = [
201     imem_error : FNONE;
202     1: imem_ifun;
203 ];
204
205 # Is instruction valid?
206 bool instr_valid = f_icode in
207     { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL, IIADDL,
208       ILEAVE,
209       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
210
211 # Determine status code for fetched instruction
212 int f_stat = [

```

```

212     imem_error: SADR;
213     !instr_valid : SINS;
214     f_icode == IHALT : SHLT;
215     1 : SAOK;
216 ];
217
218 # Does fetched instruction require a regid byte?
219 bool need_regids =
220     f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIADDL, ILEAVE,
221                 IIRMOVL, IRMMOVL, IMRMOVL };
222
223 # Does fetched instruction require a constant word?
224 bool need_valC =
225     f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
226
227 # Predict next value of PC
228 int f_predPC = [
229     f_icode in { IJXX, ICALL } : f_valC;
230     1 : f_valP;
231 ];
232
233 ##### Decode Stage #####
234
235
236 ## What register should be used as the A source?
237 int d_srcA = [
238     D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
239     D_icode in { IPOPL, IRET } : RESP;
240     D_icode in { ILEAVE } : REBP;
241     1 : RNONE; # Don't need register
242 ];
243
244 ## What register should be used as the B source?
245 int d_srcB = [
246     D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : D_rB;
247     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
248     1 : RNONE; # Don't need register
249 ];
250
251 ## What register should be used as the E destination?
252 int d_dstE = [
253     D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL } : D_rB;
254     D_icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
255     1 : RNONE; # Don't write any register
256 ];
257
258 ## What register should be used as the M destination?
259 int d_dstM = [
260     D_icode in { IMRMOVL, IPOPL } : D_rA;
261     D_icode in { ILEAVE } : REBP;
262     1 : RNONE; # Don't write any register
263 ];
264
265 ## What should be the A value?
266 ## Forward into decode stage for valA
267 int d_valA = [
268     D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC

```

```

269     d_srcA == e_dstE : e_valE;    # Forward valE from execute
270     d_srcA == M_dstM : m_valM;    # Forward valM from memory
271     d_srcA == M_dstE : M_valE;    # Forward valE from memory
272     d_srcA == W_dstM : W_valM;    # Forward valM from write back
273     d_srcA == W_dstE : W_valE;    # Forward valE from write back
274     1 : d_rvalA; # Use value read from register file
275 ];
276
277 int d_valB = [
278     d_srcB == e_dstE : e_valE;    # Forward valE from execute
279     d_srcB == M_dstM : m_valM;    # Forward valM from memory
280     d_srcB == M_dstE : M_valE;    # Forward valE from memory
281     d_srcB == W_dstM : W_valM;    # Forward valM from write back
282     d_srcB == W_dstE : W_valE;    # Forward valE from write back
283     1 : d_rvalB; # Use value read from register file
284 ];
285
286 ##### Execute Stage #####
287
288 ## Select input A to ALU
289 int aluA = [
290     E_icode in { IRRMOVL, IOPL, ILEAVE } : E_valA;
291     E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
292     E_icode in { ICALL, IPUSHL } : -4;
293     E_icode in { IRET, IPOPL } : 4;
294     # Other instructions don't need ALU
295 ];
296
297 ## Select input B to ALU
298 int aluB = [
299     E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
300         IPUSHL, IRET, IPOPL, IIADDL } : E_valB;
301     E_icode in { IRRMOVL, IIRMOVL } : 0;
302     E_icode in { ILEAVE } : 4;
303     # Other instructions don't need ALU
304 ];
305
306 ## Set the ALU function
307 int alufun = [
308     E_icode == IOPL : E_ifun;
309     1 : ALUADD;
310 ];
311
312 ## Should the condition codes be updated?
313 bool set_cc = (E_icode in { IOPL, IIADDL } ) &&
314     # State changes only during normal operation
315     !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS,
316         SHLT };
317
318 ## Generate valA in execute stage
319 int e_valA = E_valA;    # Pass valA through stage
320
321 ## Set dstE to RNONE in event of not-taken conditional move
322 int e_dstE = [
323     E_icode == IRRMOVL && !e_Cnd : RNONE;
324     1 : E_dstE;
325 ];

```

```

325 ##### Memory Stage #####
326
327
328 ## Select memory address
329 int mem_addr = [
330     M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
331     M_icode in { IPOPL, IRET } : M_valA;
332     M_icode in { ILEAVE } : M_valA;
333     # Other instructions don't need address
334 ];
335
336 ## Set read control signal
337 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET, ILEAVE };
338
339 ## Set write control signal
340 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
341
342 /* $begin pipe-m_stat-hcl */
343 ## Update the status
344 int m_stat = [
345     dmem_error : SADR;
346     1 : M_stat;
347 ];
348 /* $end pipe-m_stat-hcl */
349
350 ## Set E port register ID
351 int w_dstE = W_dstE;
352
353 ## Set E port value
354 int w_valE = W_valE;
355
356 ## Set M port register ID
357 int w_dstM = W_dstM;
358
359 ## Set M port value
360 int w_valM = W_valM;
361
362 ## Update processor status
363 int Stat = [
364     W_stat == SBUB : SAOK;
365     1 : W_stat;
366 ];
367
368 ##### Pipeline Register Control #####
369
370 # Should I stall or inject a bubble into Pipeline Register F?
371 # At most one of these can be true.
372 bool F_bubble = 0;
373 bool F_stall =
374     # Conditions for a load/use hazard
375     E_icode in { IMRMOVL, IPOPL, ILEAVE } && #Dst value
376     generated after M stage
377     E_dstM in { d_srcA, d_srcB } || #but D needs the registers
378     # Stalling at fetch while ret passes through pipeline
379     IRET in { D_icode, E_icode, M_icode };
380
381 # Should I stall or inject a bubble into Pipeline Register D?

```

```

381 # At most one of these can be true.
382 bool D_stall =
383     # Conditions for a load/use hazard
384     E_icode in { IMRMOVL, IPOPL, ILEAVE } &&    #Dst value
        generated after M stage
385     E_dstM in { d_srcA, d_srcB };    #but E needs the registers
386
387 bool D_bubble =
388     # Mispredicted branch
389     (E_icode == IJXX && !e_Cnd) ||
390     # Stalling at fetch while ret passes through pipeline
391     # but not condition for a load/use hazard
392     !(E_icode in { IMRMOVL, IPOPL, ILEAVE } && E_dstM in { d_srcA,
        d_srcB }) &&
393     IRET in { D_icode, E_icode, M_icode };
394
395 # Should I stall or inject a bubble into Pipeline Register E?
396 # At most one of these can be true.
397 bool E_stall = 0;
398 bool E_bubble =
399     # Mispredicted branch
400     (E_icode == IJXX && !e_Cnd) ||
401     # Conditions for a load/use hazard
402     E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
403     E_dstM in { d_srcA, d_srcB };
404
405 # Should I stall or inject a bubble into Pipeline Register M?
406 # At most one of these can be true.
407 bool M_stall = 0;
408 # Start injecting bubbles as soon as exception passes through
    memory stage
409 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR
    , SINS, SHLT };
410
411 # Should I stall or inject a bubble into Pipeline Register W?
412 bool W_stall = W_stat in { SADR, SINS, SHLT };
413 bool W_bubble = 0;
414 /* $end pipe-all-hcl */
415

```

2.3.3 Evaluation

Now, we are going to evaluate the result of part C.

1. Firstly, as shown in the figure below, our modifications do not change the correctness of existing instructions.

```

File Edit View Search Terminal Help
59      OK
60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test

```

- Secondly, as shown in the figure below, the ncopy file is 948bytes, less than the required 1000bytes.

```

lingzx@ubuntu:~$ (cd /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe; ./check-len.pl < ncopy)
ncopy length = 948 bytes
lingzx@ubuntu:~$

```

- Thirdly, as shown in the figure below, our implementation of iaddl and leave has survived all ISA checks.

```

lingzx@ubuntu: ~
File Edit View Search Terminal Help
lingzx@ubuntu:~$ (cd /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/ptest; ./ptest.pl -i /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim TFLAGS=-ll)
./optest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -i /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -t
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim
All 59 ISA Checks Succeed
./jtest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -i /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -t
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim
All 96 ISA Checks Succeed
./ctest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -i /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -t
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim
All 22 ISA Checks Succeed
./htest.pl -s /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -i /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim -t
Simulating with /home/lingzx/Desktop/ComputerArchitecture/prj1_2/project1-handout/sim/pipe/psim
All 870 ISA Checks Succeed
lingzx@ubuntu:~$

```

- Fourthly, as shown in the figure above, our ncopy function achieves an average CPE of 9.89, less than 10.0, which means ncopy performs very well. As expected, the larger the number of elements is, the better the function performs. That is because when there are only several elements, loop unrolling and loose ends increase branch instructions; however, when the number is larger, the reduction of branch instructions caused by loop unrolling becomes remarkable.

```

44      344      7.82
45      351      7.80
46      361      7.85
47      368      7.83
48      368      7.67
49      375      7.65
50      385      7.70
51      392      7.69
52      398      7.65
53      405      7.64
54      415      7.69
55      422      7.67
56      424      7.57
57      431      7.56
58      441      7.60
59      448      7.59
60      454      7.57
61      461      7.56
62      471      7.60
63      478      7.59
64      478      7.47
Average CPE 9.89
Score 60.0/60.0
lingzx@ubuntu:~/Desktop/ComputerArchiterture/prj1_2/project1-handout/sin/pipe$

```

3 Conclusion

3.1 Problems

There are many obstacles in the project. And we are going to share some here.

1. Firstly, adjustment to new languages. You can never image how we feel when starting the project. It is just like language bombing. Both Y86 and HCL are new to us. And the related resources on the Internet are so poor that we have no idea how to start. However, after days of searching and thinking, we gradually adjust ourselves to the new languages. Y86 is similar to but much simpler than X86 and MIPS familiar to us. As for HCL, though more complex, the project does not require us to fully master it. It is enough if we just known how to make the logic clear. Besides, thanks to CSAPP, we can acquire a lot of related materials in the book.
2. Secondly, debugging assembly codes run by pipeline processor. It is amazingly difficult to debug our codes running by a pipeline processor. Just as the pipeline architecture is complex, the procedure of debugging is complex, too. The codes are not finished line by line. An instruction is finished in the M or WB stage and the next several instructions are already on the way. However, this does not both us in the end. Gradually, we learn to examine the contents of states and inputs of different stages and this exactly deepens our understanding of pipeline processors.
3. Thirdly, understanding of stack pointer. This is exactly a detail problem. In this semester, we have read a lot of assembly codes but wrote little. The stack is used when passing arguments, saving registers and calling functions. As written in the given Y86 code examples, the convention is

that, on entering a function, we save the base pointer(`ebp`), and copy the stack pointer(`esp`) to the base pointer. It takes us a long time to figure out why the address of the first argument is `ebp+8`. And finally, by carefully observing the contents of the stack, we found when `esp` is copied to `ebp`, the stack is pushed twice: one is to save the return address and the other is to save `ebp`. Also, on the same problem, we spend a long time to debug our `rsum` function just because we pass an argument when calling the function by pushing the stack BUT forgot to pop the stack to restore the stack pointer after the function is returned.

3.2 Achievements

1. Firstly, it is great to see we have achieved an average CPE less than 10.0. This a result of our careful design of our logic and implementation. And all of us three members have contributed a lot in the process.
2. Secondly, we have successfully implemented `leave` instruction. `Leave` is more complicated than `iaddl`, especially in the pipeline architecture because it involves necessary load-read hazard, but we have implemented it.