

CS 359, COMPUTER ARCHITECTURE, YANYAN SHEN, SPRING 2017

Project2: Optimizing the Performance of a Pipelined Processor

GROUP LEADER: LI MINCHAO 515030910361

GROUP MEMBER: WANG CHENYANG 515030910383

GROUP MEMBER: QIANG ZHIWEN 515030910367

CONTACT EMAIL: marshallee413lmc@gmail.com

1 Preknowledge

I have read the CSAPP book and concluded several basic points which are essential to solving the problems in Part A, B, and C.

- There are 8 registers in the Y86 system

%eax, %ecx, %edx, %ebx

%esi, %edi, %esp, %ebp

Each of these registers stores a word. Among then register *%esp* is used as a stack pointer by the push, pop, call, and return instructions.

- There are three single-bit condition codes, ZF, SF, and OF, storing information about the effect of the most recent arithmetic or logical instruction.
- The Y86 instruction set is largely a subset of the IA32 instruction set but it still have some differences. The picture below shows the Y86 instruction set.

Byte	0	1	2	3	4	5
halt	0	0				
nop	1	0				
rrmovl rA , rB	2	0	rA	rB		
irmovl V , rB	3	0	F	rB	V	
rmmovl rA , D(rB)	4	0	rA	rB	D	
mrmmovl D(rB) , rA	5	0	rA	rB	D	
OPl rA , rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
cmovXX rA , rB	2	fn	rA	rB		
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	F		
popl rA	B	0	rA	F		

- In the Y86 system, the stack starts at a certain address and grows toward lower addresses, which prevents space conflict.

2 Part A

2.1 Description

- The program **sum.y**s was used to sum linked list elements iteratively. We are supposed to add the sum of a list from head to tail using the Y86 coding rules.
- The program **rsum.y**s is similar to the **sum.y**s, except it sums linked list elements recursively. We are supposed to add the sum of a list from head to tail using the Y86 coding rules.
- The program **copy.y**s is used for two purposes. First it copies a block of words from one part of memory to another area of memory. Second it computes the checksum (Xor) of all the words copied.

2.2 Solution

sum.y

```
1 # Initial code
2 irmovl Stack,%esp
3 rrmovl %esp,%ebp
4 irmovl ele1, %edx
5 #pushl %edx
6 call sum_list
7 halt
8
9 # Sample linked list
10 .align 4
11 ele1:
12 .long 0x00a
13 .long ele2
14 ele2:
15 .long 0x0b0
16 .long ele3
17 ele3:
18 .long 0xc00
19 .long 0
20
21 sum_list:
22 pushl %ebp                # Save %ebp
23 xorl %eax,%eax           # val = 0
24 rrmovl %esp,%ebp        # Set frame ptr
25 #pushl %edx
26 #mrmovl 8(%ebp),%edx      # Get ls
27 andl %edx,%edx           # ls == 0?
28 je L4                   # Yes, goto done
29
30 L5:                      # Loop:
31 mrmovl (%edx),%esi       # t = ls->val
32 addl %esi,%eax           # val += t
33 mrmovl 4(%edx),%edx      # ls = ls->next
34 andl %edx,%edx           # ls == 0?
```

```

35 jne L5                # No, goto done
36
37 L4:                   # Done:
38 rrmovl %ebp,%esp      # Restore %esp
39 popl %ebp             # Restore %ebp
40 ret                   # Return
41
42 .pos 0x100
43 Stack:

```

rsum.ys

```

1 # Execution begins at address 0
2 .pos 0
3 init:                irmovl Stack, %esp
4 irmovl Stack, %ebp
5 jmp Main
6
7 # Sample linked list
8 .align 4
9 ele1:                .long 0x00a
10 .long ele2
11 ele2:                .long 0x0b0
12 .long ele3
13 ele3:                .long 0xc00
14 .long 0
15
16 Main:                irmovl ele1, %edx
17 pushl %ebp
18 rrmovl %esp, %ebp
19 pushl %edx
20 call rsum_list
21 rrmovl %ebp, %esp
22 popl %ebp
23 halt
24
25 # rsum_list - Recursive version of sum_list
26 # int rsum_list(list_ptr ls)
27 rsum_list:           pushl                %ebp
28 rrmovl                %esp,%ebp
29 mrmovl                0x8(%ebp),%edx    # ls
30 xorl                  %eax,%eax        # val=0
31 pushl                 %ebx             # save %ebx
32 andl                  %edx,%edx        # ls==0?
33 je                    End              # if so, gotoEnd
34 mrmovl                (%edx),%ebx      # ls->val
35 mrmovl                0x4(%edx),%ecx   # ls->next
36 pushl                 %ecx
37 # push ls->next as the first parameter
38 call                  rsum_list
39 # call rsum_list by recursion

```

```

40 addl          %ebx,%eax          # val+=ls->val
41 End:          mrmovl 0xffffffff(%ebp),%ebx
42 # restore %ebx
43 rrmovl        %ebp, %esp
44 popl          %ebp
45 ret
46
47 .pos 0x100
48 Stack:

```

copy.y

```

1  .pos 0
2  init:  irmovl Stack, %esp
3  irmovl Stack, %ebp
4  jmp Main
5
6  .align 4
7  # Source block
8  src:
9  .long 0x00a
10 .long 0x0b0
11 .long 0xc00
12 # Destination block
13 dest:
14 .long 0x111
15 .long 0x222
16 .long 0x333
17
18 Main:  irmovl $3,%eax
19 pushl %eax
20 irmovl dest,%edx
21 pushl %edx
22 irmovl src,%ecx
23 pushl %ecx
24 call Copy
25 halt
26
27 Copy:  pushl %ebp
28 rrmovl %esp,%ebp
29 mrmovl 8(%ebp),%ecx    #ecx = src
30 mrmovl 12(%ebp),%ebx   #ebx = dest
31 mrmovl 16(%ebp),%edx   #edx = len
32 irmovl $0,%eax        #result = 0
33 andl %edx,%edx
34 je End
35 Loop:  mrmovl (%ecx),%esi    #get *src
36 rrmovl %esi,(%ebx)        #scr = dest
37 xorl %esi,%eax          #result ^= src
38 irmovl $4,%edi          #set %edi to 4
39 addl %edi,%ecx          #+4

```

```

40 | addl %edi,%ebx          #+4
41 | irmovl $-1,%edi        #set %edi to -1
42 | addl %edi,%edx          #len - 1
43 | jne      Loop          #Stop when 0
44 |
45 |
46 | End:      popl %ebp
47 | rrmovl %ebp, %esp
48 | popl %ebp
49 | ret
50 |
51 | .pos 0x100
52 | Stack:

```

2.3 Analysis

2.3.1 sum.js

- For the program **sum.js**, the line 1 ~ 19 and line 42 ~ 43 are the preparatory work and do not need further explanation. For the main part, which is the `sum.js`, first we store the `%ebp` part to stack, then we set `%eax` to zero as the initial sum value, `%edx` as the initial list position, then if the list has reached its end, we jump to state L4, which restore the value. If not, we goto the LOOP L5 part.
- In the L5 part, first `%eax+ = %esi` to add to the sum, then `(%edx) + 4` to goto the list's next elements. After doing that, again we judge whether the list has reached its end and the condition is exactly the same.
- Based the outcome of this program, `%eax` stores the overall sum value, which is `0xcba`, `%ebp` get popped and get its initial value, which is the `0x100`, `%esi` stores the last elements of the list, which is `%0xc00`, which proves that our program is correct.

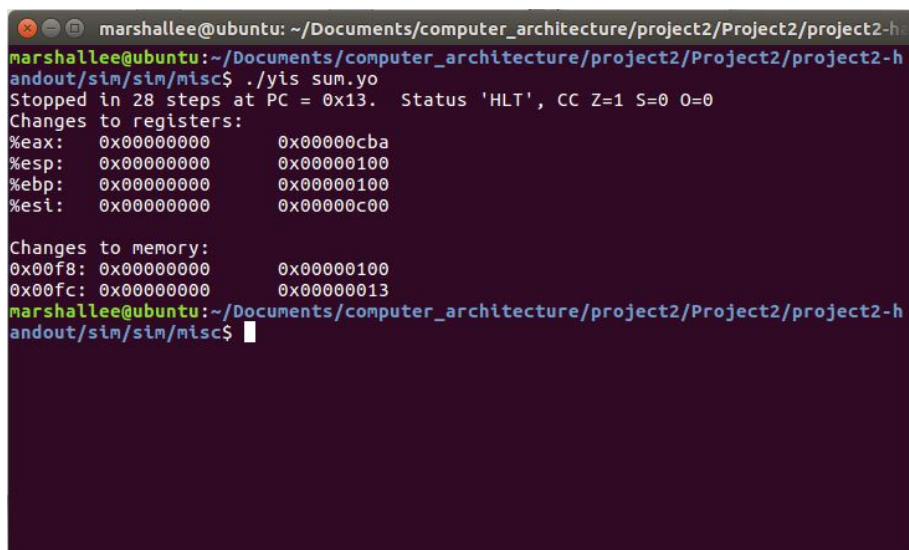
2.3.2 rsum.js

- For the program **rsum.js**, the line 1 ~ 14 and line 47 ~ 48 are the preparatory work and do not need further explanation. For the main part, first we let `%edx` stores the `el`, which is the beginning, then we save `%edx` and call `rsum_list`.
- For the `rsum_list` part(line27 ~ 45), which is the recursive version of the `sum_list`, first we use `%edx` to get starting address, `%eax` to get the initial result which is 0, then we save `%ebx` and compare whether `%edx` is zero or not. If so, goto `end`, else, goto the next part, which is the recursive part. It is worth noting that in line 39 the constant number `0xffffffffc` is `-4` and we use this operation to restore `%ebx`, and in line 17, we push `%ebp`, in line 44, we pop `%ebp`, with these two operations we can get the old version of `%ebp` which is key to our recursive part.
- Based the outcome of this program, `%ebp` pop out and its initial value is `0x100`, `%eax` stores the overall sum value `0xcba`, which proves that our program is correct.

2.3.3 copy.yo

- For the program **copy.yo**, the line 1 ~ 16 and line 51 ~ 52 are the preparatory work and do not need further explanation. For the main part(line 18 ~ 25), first we change *%eax* to 3, *%edx* to the *dest*, *%ecx* to the *src* and store the value to the stack, then we call Copy function and halt.
- For the Copy function(line 28 ~ 43), first we let *%ebp* store *%esp*, *%ecx* store *src*, *%ebx* store *dest*, *%edx* store the length, *%eax* stores the initial value of the result which is 0, then we compare whether the copy function has reached its end, is so, goto the End part, else, goto the Loop part.
- For the Loop part, first we get the address of the *scr*, then we begin the copy process, while we add the *src* and *dest* by 4 to move to the next address, in the meantime we let *len - 1* to serve as flag to decide when to stop.
- Based the outcome of this program, register *%eax* do the *xorl* instruction with each value and then add to *%eax*, so the last value of *%eax* is *0xcba*, also the address *0x0020* value is changed from *0x111* to *0xa*, the address *0x0024* value is changed from *0x222* to *0xb0*, the address *0x0028* value is changed from *0x333* to *0xc00*, which proves that our program is correct.

2.4 Outcome



```
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-h
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$ ./yis sum.yo
Stopped in 28 steps at PC = 0x13. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%esp: 0x00000000      0x00000100
%ebp: 0x00000000      0x00000100
%esi: 0x00000000      0x00000c00
Changes to memory:
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000013
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$
```

```

marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-h
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$ ./yis rsum.yo
Stopped in 70 steps at PC = 0x41. Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%esp: 0x00000000      0x00000100
%ebp: 0x00000000      0x00000100

Changes to memory:
0x00bc: 0x00000000      0x00000c00
0x00c0: 0x00000000      0x00000d00
0x00c4: 0x00000000      0x0000006a
0x00cc: 0x00000000      0x000000b0
0x00d0: 0x00000000      0x000000e0
0x00d4: 0x00000000      0x0000006a
0x00d8: 0x00000000      0x00000024
0x00dc: 0x00000000      0x0000000a
0x00e0: 0x00000000      0x000000f0
0x00e4: 0x00000000      0x0000006a
0x00e8: 0x00000000      0x0000001c
0x00f0: 0x00000000      0x000000fc
0x00f4: 0x00000000      0x0000003d
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000100
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$

```

```

marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-h
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$ ./yis copy.yo
Stopped in 10000 steps at PC = 0x32. Status 'AOK', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000003
%ecx: 0x00000000      0x00000020
%ebx: 0x00000000      0x0000002c
%esp: 0x00000000      0x00000100
%ebp: 0x00000000      0x00000100
%esi: 0x00000000      0x00000c00
%edi: 0x00000000      0xffffffff

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x00ec: 0x00000000      0x00000100
0x00f0: 0x00000000      0x00000049
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000020
0x00fc: 0x00000000      0x00000003
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-h
andout/sim/sim/misc$

```


3 Part B

3.1 Description

In this part, we are asked to add a new instruction *iaddl* to the SEQ processor, this instruction is meant to add a constant to a register.

3.2 Solution

Stage	iaddl V, rB
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_4[PC+2]$ $valP \leftarrow PC+6$
Decode	$valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB \text{ ADD } valC$
Memory	
Write back	$R[rB] \leftarrow valE$
PC update	$PC \leftarrow valP$

Since the code in seq-full.hcl is quite long, I only list the part where we have made changes.

Fetch Stage

```
1  int main(void)
2  {
3      struct sigaction handler;
4      handler.sa_handler = handler.SIGINT;
5      sigaction(SIGINT, &handler, NULL);
6      //strcpy(buffer, "\nCaught Control C\n");
7
8
9      while(1)
10     {
11         printf("COMMAND->");
12         setup(inputBuffer, args, &background);
13         if(!strlen(inputBuffer)) continue;
14         fpid = fork();
15         switch(fpid)
16         {
17             case -1: perror("fork_failed"); exit(1);
18             case 0 : execvp(args[0], args); perror("execvp_failed");
19             default:
20                 if(background) while(wait(&exitstatus) != fpid)
21                 }
22             inputBuffer[0] = 0;
23         }
24         return 0;
25     }
```

3.3 Analysis

It can be implemented by first using *irmovl* instruction to let the register contains the constant number, then we can use *addl* instruction to add the constant number to the destination register. Since there are roughly four stages in the Y86 instruction set, we will fully discuss this part regarding each stages.

- For the Fetch Stage, first we should add the *IIADDL* instruction to the *instr_valid*, then we should add the *IIADDL* instruction to the *need_regids* set, indicating that we should register to do this operation. Finally since we need a constant to do the addition, we need to add the *IIADDL* instruction to the *need_valc* set.
- For the Decode Stage, first we need to add the *IIADDL* instruction to the *srcB* set, indicating that we put the register value of *rB* in this part, then we need to add the *IIADDL* instruction to the *dstE* set, indicating that we store the value in the destination E, which is to the same register of *rB*.
- For the Execute Stage, first we should add the *IIADDL* instruction to the *aluA* part, indicating that the ALU operation need the *valC*, which is the constant value, then we should add the *IIADDL* instruction to the *aluB* part, indicating that the ALU operation need the *valB*, finally we should add the *IIADDL* instruction to the *set_cc* part, indicating that this instruction may lead the flag register to change.
- For the Memory Stage, since this instruction is to add a constant number to a register, so the Memory Stage don't have to change.

3.4 Outcome

```
marshalllee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq
Y86 Processor: seq-full.hcl
112 bytes of code read
IF: Fetched irmovl at 0x0. ra=----, rb=Kesp, valC = 0x100
IF: Fetched irmovl at 0x6. ra=----, rb=Kebp, valC = 0x100
IF: Fetched jmp at 0xc. ra=----, rb=----, valC = 0x24
IF: Fetched irmovl at 0x24. ra=----, rb=Keax, valC = 0x4
IF: Fetched pushl at 0x2a. ra=Kesp, rb=----, valC = 0x0
Wrote 0x4 to address 0xf4
IF: Fetched irmovl at 0x2c. ra=----, rb=Kedx, valC = 0x14
IF: Fetched pushl at 0x32. ra=Kedx, rb=----, valC = 0x0
Wrote 0x14 to address 0xf8
IF: Fetched call at 0x34. ra=----, rb=----, valC = 0x3a
Wrote 0x39 to address 0xf4
IF: Fetched pushl at 0x3a. ra=Kesp, rb=----, valC = 0x0
Wrote 0x100 to address 0xf0
IF: Fetched rrmovl at 0x3c. ra=Kesp, rb=Kebp, valC = 0x0
IF: Fetched rrmovl at 0x3e. ra=Kecx, rb=Kebp, valC = 0x8
IF: Fetched rrmovl at 0x44. ra=Kedx, rb=Kebp, valC = 0xc
IF: Fetched irmovl at 0x4a. ra=----, rb=Keax, valC = 0xc
IF: Fetched andl at 0x50. ra=Kedx, rb=Kedx, valC = 0x0
IF: Fetched je at 0x52. ra=----, rb=----, valC = 0x70
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70. ra=Kebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72. ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39. ra=----, rb=----, valC = 0x0

marshalllee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq
IF: Fetched rrmovl at 0x3c. ra=Kesp, rb=Kebp, valC = 0x0
IF: Fetched rrmovl at 0x3e. ra=Kecx, rb=Kebp, valC = 0x8
IF: Fetched rrmovl at 0x44. ra=Kedx, rb=Kebp, valC = 0xc
IF: Fetched irmovl at 0x4a. ra=----, rb=Keax, valC = 0xc
IF: Fetched andl at 0x50. ra=Kedx, rb=Kedx, valC = 0x0
IF: Fetched je at 0x52. ra=----, rb=----, valC = 0x70
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched rrmovl at 0x57. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched addl at 0x5d. ra=Kesi, rb=Kecx, valC = 0x0
IF: Fetched laddl at 0x5f. ra=----, rb=Kecx, valC = 0x4
IF: Fetched laddl at 0x65. ra=----, rb=Kedx, valC = 0xffffffff
IF: Fetched jne at 0x6b. ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70. ra=Kebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72. ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39. ra=----, rb=----, valC = 0x0
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax: 0x00000000 0x000000bcd
%ecx: 0x00000000 0x00000024
%esp: 0x00000000 0x000000f8
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x00000a00
Changed Memory State:
0x00f0: 0x00000000 0x00000100
0x00f4: 0x00000000 0x00000039
0x00f8: 0x00000000 0x00000014
0x00fc: 0x00000000 0x00000004
ISA Check Succeeds
marshalllee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq
```

4 Part C

4.1 Description

- The program **ncopy.y**s copies a len-element integer array *src* to a non-overlapping *dst*, returning the count number of positive integers contained in *src*.
- In this part, we need to minimize the running time of **ncopy.y**s as less as possible. We use 3 ways to decrease running time: decrease jump instruction, use iaddl instruction and decrease hazards.
- As we have learned in the class, the *jump* instructions (including *jmp*, *jle*, *jl*, *je*, *jne*, *jge* and *jg* in Y86 instructions). Of course we can write one loop to solve this problem, but it costs too much time. So we use some other methods to solve this problem.
- In traditional add instruction, we can only store the instant number in registers and add the values in registers, which increase the running time. The iaddl instruction can do the add instruction between instant number and register, which decrease the running time.
- If we store a value in a register and the next instruction is to use the value in this register, there must be a stall. So we use 2 registers to do it so to decrease stalls, which also decrease our running time.

4.2 Solution

ncopy.y

```
1  /* $begin ncopy-y */
2  #####
3  # ncopy.y - Copy a src block of len ints to dst.
4  # Return the number of positive ints (>0) contained in src.
5  #
6  # Include your name and ID here.
7  #
8  # Describe how and why you modified the baseline code.
9  #
10 #####
11 # Do not modify this portion
12 # Function prologue.
13 ncopy:  pushl %ebp                # Save old frame pointer
14         rrmovl %esp,%ebp          # Set up new frame pointer
15         pushl %esi                # Save callee-save regs
16         pushl %ebx
17         pushl %edi
18         mrmovl 8(%ebp),%ebx        # src
19         mrmovl 16(%ebp),%edx       # len
20         mrmovl 12(%ebp),%ecx       # dst
21 #####
22         xorl  %eax,%eax            # initialize the count to 0
23
24 #####
25 Loop8:
```

```

26 | iaddl $-8, %edx          # len = len - 8
27 | andl %edx, %edx         # to see if the len is less than 0
28 | jl Loop4
29 |
30 | mrmovl (%ebx), %esi
31 | mrmovl 4(%ebx), %edi
32 | rmmovl %esi, (%ecx)
33 | andl %esi, %esi
34 | jle StageOneOf8
35 | iaddl $1, %eax
36 |
37 | StageOneOf8:
38 | rmmovl %edi, 4(%ecx)
39 | andl %edi, %edi
40 | jle StageTwoOf8
41 | iaddl $1, %eax
42 |
43 | StageTwoOf8:
44 | mrmovl 8(%ebx), %esi
45 | mrmovl 12(%ebx), %edi
46 | rmmovl %esi, 8(%ecx)
47 | andl %esi, %esi
48 | jle StageThreeOf8
49 | iaddl $1, %eax
50 |
51 | StageThreeOf8:
52 | rmmovl %edi, 12(%ecx)
53 | andl %edi, %edi
54 | jle StageFourOf8
55 | iaddl $1, %eax
56 |
57 | StageFourOf8:
58 | mrmovl 16(%ebx), %esi
59 | mrmovl 20(%ebx), %edi
60 | rmmovl %esi, 16(%ecx)
61 | andl %esi, %esi
62 | jle StageFiveOf8
63 | iaddl $1, %eax
64 |
65 | StageFiveOf8:
66 | rmmovl %edi, 20(%ecx)
67 | andl %edi, %edi
68 | jle StageSixOf8
69 | iaddl $1, %eax
70 |
71 | StageSixOf8:
72 | mrmovl 24(%ebx), %esi
73 | mrmovl 28(%ebx), %edi
74 | rmmovl %esi, 24(%ecx)
75 | andl %esi, %esi

```

```

76 | jle StageSevenOf8
77 | iaddl $1, %eax
78 |
79 | StageSevenOf8:
80 | rmmovl %edi, 28(%ecx)
81 | andl %edi, %edi
82 | jle Forward8
83 | iaddl $1, %eax
84 |
85 | Forward8:
86 | iaddl $32, %ebx
87 | iaddl $32, %ecx
88 | jmp Loop8
89 |
90 | #####
91 | Loop4:
92 | iaddl $8, %edx
93 | iaddl $-4, %edx
94 | andl %edx, %edx
95 | jl Loop2
96 |
97 | mrmovl (%ebx), %esi
98 | mrmovl 4(%ebx), %edi
99 | rmmovl %esi, (%ecx)
100 | andl %esi, %esi
101 | jle StageOneOf4
102 | iaddl $1, %eax
103 |
104 | StageOneOf4:
105 | rmmovl %edi, 4(%ecx)
106 | andl %edi, %edi
107 | jle StageTwoOf4
108 | iaddl $1, %eax
109 |
110 | StageTwoOf4:
111 | mrmovl 8(%ebx), %esi
112 | mrmovl 12(%ebx), %edi
113 | rmmovl %esi, 8(%ecx)
114 | andl %esi, %esi
115 | jle StageThreeOf4
116 | iaddl $1, %eax
117 |
118 | StageThreeOf4:
119 | rmmovl %edi, 12(%ecx)
120 | andl %edi, %edi
121 | jle Forward4
122 | iaddl $1, %eax
123 |
124 | Forward4:
125 | iaddl $16, %ebx

```

```

126 iaddl $16, %ecx
127 iaddl $-4, %edx
128 jmp Loop2
129
130 #####
131 Loop2:
132 iaddl $4, %edx
133 iaddl $-2, %edx
134 andl %edx, %edx
135 jl Loop1
136
137 mrmovl (%ebx), %esi
138 mrmovl 4(%ebx), %edi
139 rmmovl %esi, (%ecx)
140 andl %esi, %esi
141 jle StageOneOf2
142 iaddl $1, %eax
143
144 StageOneOf2:
145 rmmovl %edi, 4(%ecx)
146 andl %edi, %edi
147 jle Forward2
148 iaddl $1, %eax
149
150 Forward2:
151 iaddl $8, %ebx
152 iaddl $8, %ecx
153 iaddl $-2, %edx
154 jmp Loop1
155
156 #####
157 Loop1:
158 iaddl $2, %edx
159 iaddl $-1, %edx
160 andl %edx, %edx
161 jl Done
162
163 mrmovl (%ebx), %esi
164 rmmovl %esi, (%ecx)
165 andl %esi, %esi
166 jle Done
167 iaddl $1, %eax
168 jmp Done
169 #####
170 # Do not modify the following section of code
171 # Function epilogue.
172 Done:
173 popl %edi           # Restore callee-save registers
174 popl %ebx
175 popl %esi

```

```

176 rrmovl %ebp, %esp
177 popl %ebp
178 ret
179 #####
180 # Keep the following label at the end of your function
181 End:
182 #/* $end ncopy-ys */

```

Fetch Stage

```

1 # Is instruction valid?
2 bool instr_valid = f_icode in
3 { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
4 IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
5
6 # Does fetched instruction require a regid byte?
7 bool need_regids =
8 f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
9 IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
10
11 # Does fetched instruction require a constant word?
12 bool need_valC =
13 f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };

```

Decode Stage

```

1 ## What register should be used as the B source?
2 int d_srcB = [
3 D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : D_rB;
4 D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
5 1 : RNONE; # Don't need register
6 ];
7
8 ## What register should be used as the E destination?
9 int d_dstE = [
10 D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL } : D_rB;
11 D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
12 1 : RNONE; # Don't write any register
13 ];

```

Execute Stage

```

1 ## Select input A to ALU
2 int aluA = [
3 E_icode in { IRRMOVL, IOPL } : E_valA;
4 E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
5 E_icode in { ICALL, IPUSHL } : -4;
6 E_icode in { IRET, IPOPL } : 4;
7 # Other instructions don't need ALU
8 ];
9
10 ## Select input B to ALU

```



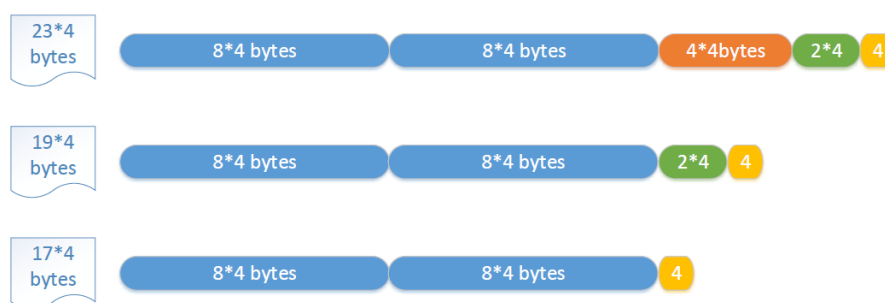
```

11 int aluB = [
12   E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
13   IPUSHL, IRET, IPOPL, IIADDL } : E_valB;
14   E_icode in { IRRMOVL, IIRMOVL } : 0;
15   # Other instructions don't need ALU
16 ];
17
18 ## Should the condition codes be updated?
19 bool set_cc = E_icode in { IOPL , IIADDL } ;

```

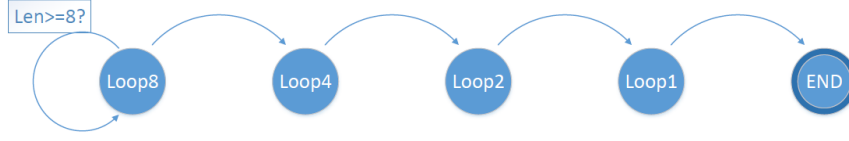
4.3 Analysis

4.3.1 ncopy.js



- We use 3 ways to decrease running time: decrease jump instruction, use iaddl instruction and decrease hazards. To explain our solution clearly, we provide the above picture to express the overview of our solution.
- Our main idea is: In each copy instruction, we copy 4 *bytes*. Firstly, we copy each 8 * 4 *bytes* in a loop until the left data needed to copy is less than 8 * 4 *bytes*. Secondly, we check if the left data is more or equal to 4 * 4 *bytes*. If so, we copy 4 * 4 *bytes* in this step. Thirdly, we check if the left data is more or equal to 2 * 4 *bytes*. If so, we copy 2 * 4 *bytes* in this step. Finally, we check if there are still 1 * 4 *bytes* left. If so, we copy these 4 *bytes* and end the problem. Since we need to return the count number of positive integers contained in *src*, we will do a judge after each copy instruction to decide if we need to increase the count by 1.
- For the instruction *iaddl*, we will explain why we use it. In traditional instructions, if we want to add an instant number and a value in register, we need to store this number in a register and add values in 2 registers, which need 2 instructions to do this. However, the *iaddl* instruction allows us to add an instant number and a value in register in just 1 instruction. This will decrease the running time.
- For the registers, we define *esi* and *edi* to store the value of contiguous 2 words (4 bytes each word). We define *ebx* to store the address of current *src* and *dst* to store the address of current *dst*. We define *edx* to store the *len* of left words needed to copy. We define *eax* to store the count of positive integers in our copied data.
- For register *esi* and *edi*, we will explain why we use 2 registers to store the copy integers. Assume we just use one register, such as *esi* to do this job. Since the next instruction is to copy the value of *esi* to *ecx*, there must be a stall. However, if we use 2 registers, there are 1 instruction using *edi* between these 2 instructions using *esi*, there will be no stall. This method can decrease the running time.

- For the Loops, we define 4 main blocks: *Loop8*, *Loop4*, *Loop2* and *Loop1*. However, only the *Loop8* is the "real" Loop which means only *Loop8* can be excuted more than 1 time. In the above picture, blue means *Loop8*, oringe means *Loop4*, green means *Loop2* and yellow means *Loop1*. We provide following picture to represent the relations among these Loop.



- We take *Loop4* for example. First we plus 4 to *edx* and check if it is less than 0. If it is, the program will jump to *Loop2*. Else, we do 4 copy instructions in *Loop4*. The register *esi* and *edi* stores the current data we need to copy. After each copy instruction, we will check if the data is a positive integer. If it is, the register *eax* will be added by 1. Else, the problem will ignore the add instruction and jump to next copy instruction.
- Assume we need to copy n words and n is a very large number. Assume the count number of positive integers contained in *src* is m . In our progress, we need $O(\log_8^n + m)$ jumps. That explains why we use this loop method to decrease the number of jump. Since if we just use 1 loop to do this job, we need $O(n)$ jumps.

4.3.2 pipe-full.hcl

The analysis in "pipe-full.hcl" is very similar to part B.

Stage	iaddl V, rB
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_4[PC+2]$ $valP \leftarrow PC+6$
Decode	$valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB \text{ ADD } valC$
Memory	
Write back	$R[rB] \leftarrow valE$
PC update	$PC \leftarrow valP$

- For the Fetch Stage, first we should add the *IIADDL* instruction to the *instr_valid*, then we should add the *IIADDL* instruction to the *need_regids* set, indicating that we need register to do this operation. Finally since we need a constant to do the addition, we need to add the *IIADDL* instruction to the *need_valC* set.
- For the Decode Stage, first we need to add the *IIADDL* instruction to the *d_srcB* set, indicating that we put the register value of *rB* in this part, then we need to add the *IIADDL* instruction to the *d_dstE* set, indicating that we store the value in the destination E, which is to the same register of *rB*.
- For the Execute Stage, first we should add the *IIADDL* instruction to the *aluA* part, indicating that the ALU operation need the *valC*, which is the constant value, then we should add the *IIADDL* instruction to the *aluB* part, indicating that the ALU operation need the

valB, finally we should add the *IIADDL* instruction to the *set_cc* part, indicating that this instruction may lead the flag register to change.

- For the Memory Stage, since this instruction is to add a constant number to a register, so the Memory Stage don't have to change.

4.4 Outcome

```

marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe$ ./benchmark.pl
ncopy
0      43
1      50      50.00
2      68      30.00
3      67      22.33
4      71      17.75
5      78      15.00
6      88      14.07
7      95      13.57
8      95      11.88
9     102      11.33
10     112      11.20
11     119      10.82
12     123      10.25
13     130      10.00
14     140      10.00
15     147      9.80
16     147      9.29
17     154      9.06
18     164      9.11
19     171      9.00
20     175      8.75
21     182      8.67
22     192      8.73
23     199      8.65
24     199      8.29
25     206      8.24
26     216      8.31
27     224      8.30
28     227      8.11
29     234      8.07
30     244      8.13
31     252      8.13
32     251      7.84
33     258      7.82
34     268      7.88
35     276      7.89
36     279      7.75
37     286      7.73
38     296      7.79
39     304      7.79
40     303      7.58
41     310      7.56
42     320      7.62

```

```

marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe$ ./benchmark.pl
23     199      8.65
24     199      8.29
25     206      8.24
26     216      8.31
27     224      8.30
28     227      8.11
29     234      8.07
30     244      8.13
31     252      8.13
32     251      7.84
33     258      7.82
34     268      7.88
35     276      7.89
36     279      7.75
37     286      7.73
38     296      7.79
39     304      7.79
40     303      7.58
41     310      7.56
42     320      7.62
43     328      7.63
44     331      7.52
45     338      7.51
46     348      7.57
47     356      7.57
48     355      7.40
49     362      7.39
50     372      7.44
51     380      7.45
52     383      7.37
53     391      7.38
54     400      7.41
55     407      7.40
56     407      7.27
57     414      7.26
58     424      7.31
59     431      7.31
60     435      7.25
61     443      7.20
62     452      7.29
63     459      7.29
64     459      7.17
Average CPE 9.90
Score 60.0/60.0
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe$

```

```
marshallee@ubuntu:~/Documents/computer_architecture/project2/project2-handout/sim/sim/pipe$ ./correctness.pl
Simulating with instruction set simulator yls
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
```

```
marshallee@ubuntu:~/Documents/computer_architecture/project2/project2-handout/sim/sim/pipe$ ./correctness.pl
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
marshallee@ubuntu:~/Documents/computer_architecture/project2/project2-handout/sim/sim/pipe$
```

5 Task Assignments

Li Minchao write codes for partA, partB and partC.

Wang Chenyang write description and report for partC and integrate the report.

Qiang Zhiwen write description and report for partA and partB and write preknowledge of report.