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

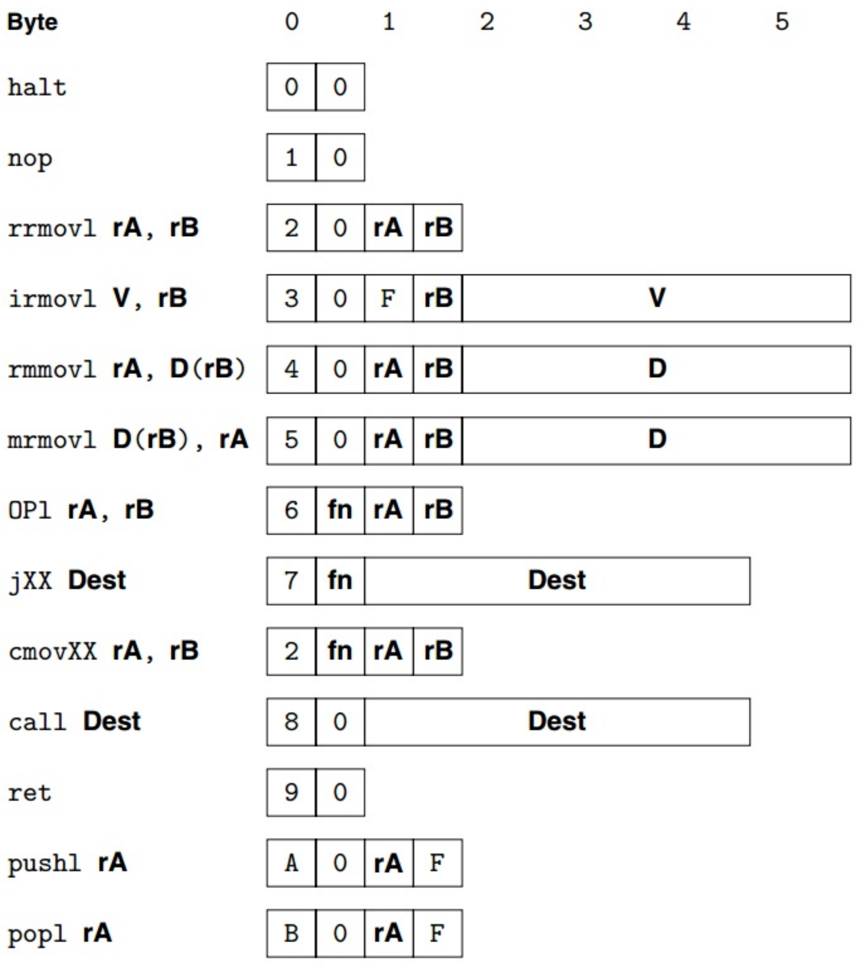
$$\%eax, \%ecx, \%edx, \%ebx$$

$$\%esi, \%edi, \%esp, \%ebp$$

  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			
`mrmovl` **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.ys** 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 codeing rules.
- The program **rsum.ys** is similar to the **sum.ys**, except it sums linked list elements recursively . We are supposed to add the sum of a list from head to tail using the Y86 codeing rules.
- The program **copy.ys** 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.ys

```
# Initial code
irmovl Stack,%esp
rrmovl %esp,%ebp
irmovl ele1 , %edx
#pushl %edx
call sum_list
halt

# Sample linked list
.align 4
ele1:
.long 0x00a
.long ele2
ele2:
.long 0x0b0
.long ele3
ele3:
.long 0xc00
.long 0

sum_list:
pushl %ebp                # Save %ebp
xorl %eax,%eax            # val = 0
rrmovl %esp,%ebp          # Set frame ptr
#pushl %edx
#mrmovl 8(%ebp),%edx      # Get ls
andl %edx,%edx            # ls == 0?
je L4                     # Yes, goto done

L5:                       # Loop:
mrmovl (%edx),%esi        # t = ls->val
addl %esi,%eax            # val += t
mrmovl 4(%edx),%edx       # ls = ls->next
andl %edx,%edx            # ls == 0?
```

```
jne L5                  # No, goto done

L4:                     # Done:
rrmovl %ebp,%esp        # Restore %esp
popl %ebp               # Restore %ebp
ret                     # Return

.pos 0x100
Stack:
```

rsum.ys

```
# Execution begins at address 0
.pos 0
init:           irmovl Stack, %esp
irmovl Stack, %ebp
jmp Main

# Sample linked list
.align 4
ele1:           .long 0x00a
.long ele2
ele2:           .long 0x0b0
.long ele3
ele3:           .long 0xc00
.long 0

Main:           irmovl ele1, %edx
pushl %ebp
rrmovl %esp, %ebp
pushl %edx
call rsum_list
rrmovl %ebp, %esp
popl %ebp
halt

# rsum_list - Recursive version of sum_list
# int rsum_list(list_ptr ls)
rsum_list:      pushl           %ebp
rrmovl          %esp,%ebp
mrmovl          0x8(%ebp),%edx  # ls
xorl            %eax,%eax       # val=0
pushl           %ebx            # save %ebx
andl            %edx,%edx       # ls==0?
je              End             # if so, gotoEnd
mrmovl          (%edx),%ebx     # ls->val
mrmovl          0x4(%edx),%ecx  # ls->next
pushl           %ecx
# push ls->next as the first parameter
call            rsum_list
# call rsum_list by recursion
```

```
addl              %ebx,%eax         # val+=ls->val
End:              mrmovl   0xfffffffc(%ebp),%ebx
# restore %ebx
rrmovl            %ebp, %esp
popl              %ebp
ret

.pos     0x100
Stack:
```

copy.ys

```
.pos 0
init:    irmovl Stack, %esp
irmovl Stack, %ebp
jmp Main

.align 4
# Source block
src:
.long 0x00a
.long 0x0b0
.long 0xc00
# Destination block
dest:
.long 0x111
.long 0x222
.long 0x333

Main:    irmovl $3,%eax
pushl %eax
irmovl dest,%edx
pushl %edx
irmovl src,%ecx
pushl %ecx
call Copy
halt

Copy:    pushl %ebp
rrmovl %esp,%ebp
mrmovl 8(%ebp),%ecx       #ecx = src
mrmovl 12(%ebp),%ebx      #ebx = dest
mrmovl 16(%ebp),%edx      #edx = len
irmovl $0,%eax            #result = 0
andl %edx,%edx
je End
Loop:    mrmovl (%ecx),%esi       #get *src
rmmovl %esi,(%ebx)        #scr = dest
xorl %esi,%eax            #result ^= src
irmovl $4,%edi            #set %edi to 4
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.ys

- For the program **sum.ys**, the line 1 $\sim$ 19 and line 42 $\sim$ 43 are the preparatory work and do not need further explanation. For the main part, which is the sum.ys, first we store the $\%ebp$ part to stack, then we set $\%eax$ to zero as the inital sum value, $\%edx$ as the inital 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 inital 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.ys

- For the program **rsum.ys**, the line 1 $\sim$ 14 and line 47 $\sim$ 48 are the preparatory work and do not need further explanation. For the main part, first we let $\%edx$ stores the elel, which is the beginning, then we save $\%edx$ and call *rsum_list*.

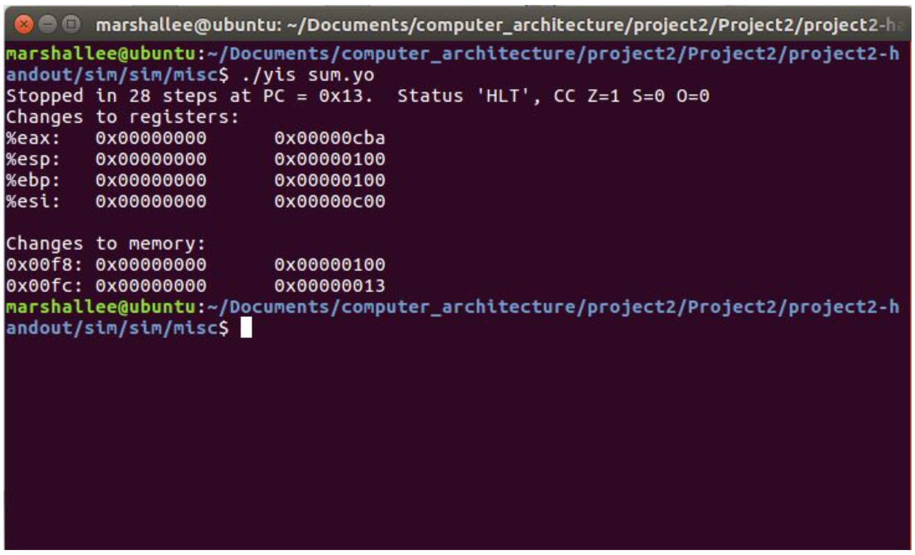
- For the *rsum_list* part(line27 $\sim$ 45),which is the recursive version of the *sum_list*, first we use $\%edx$ to get starting adddress, $\%eax$ to get the inital result which is 0, then we save $\%ebx$ and compare whether $\%edx$ is zero or not. If so, goto *end*, else, goto the next part, which is the recursive part. It is worth noting that in line 39 the constant number $0xfffffffc$ 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 inital value is $0x100$, $\%eax$ stores the overall sum value $0xcba$, which proves that our program is correct.

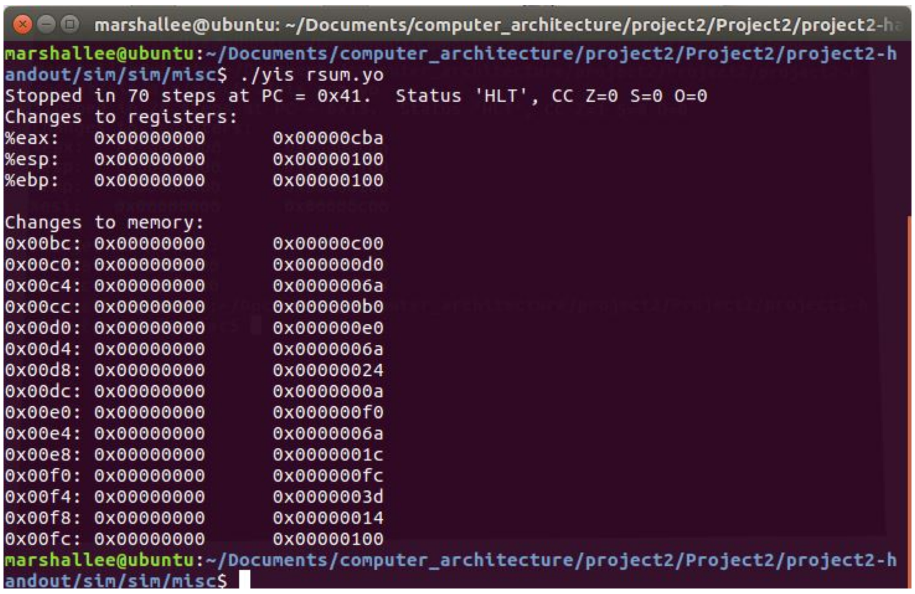
2.3.3 copy.ys

- For the program **copy.ys**, 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 inital 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.

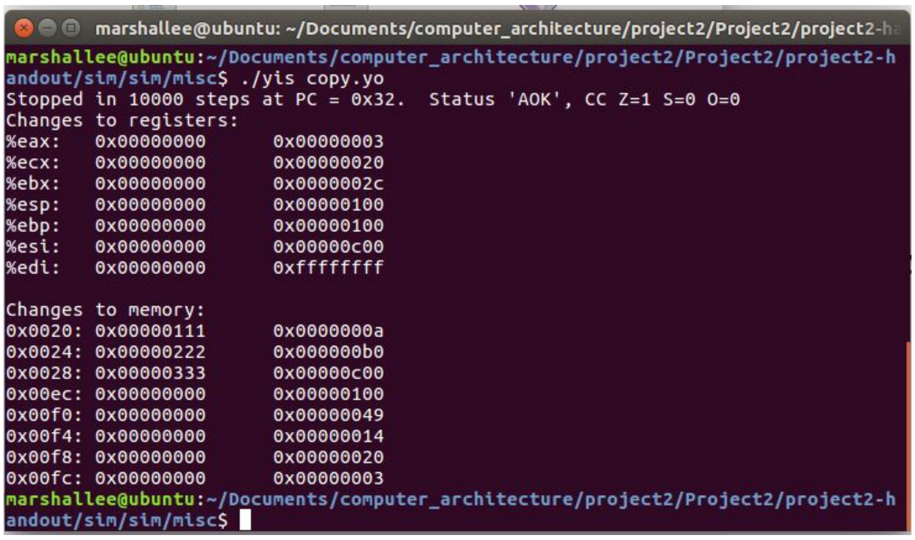
2.4 Outcome



```
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-ha
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      0x000000d0
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-ha
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 *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

```
bool instr_valid = icode in
{ INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
#show that it is valid

bool need_regids =
icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
#show that we need the register

bool need_valC =
icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
#show that we need a constant number
```

Decode Stage

```
int srcB = [
icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE;
];
#show that we put the register value of srcB

int dstE = [
icode in { IRRMOVL } && Cnd : rB;
icode in { IIRMOVL, IOPL, IIADDL} : rB;
icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE;
];
#show that we store the value in the destination E
```

Execute Stage

```
int aluA = [
icode in { IRRMOVL, IOPL } : valA;
icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
icode in { ICALL, IPUSHL } : -4;
icode in { IRET, IPOPL } : 4;
];
#show that the ALU operation need the valC

int aluB = [
icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
IPUSHL, IRET, IPOPL, IIADDL } : valB;
icode in { IRRMOVL, IIRMOVL } : 0;
];
#show that the ALU operation need the valB

bool set_cc = icode in { IOPL, IIADDL };
#show that this instruction may lead the flag register to change.
```

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 reagarding each stages.

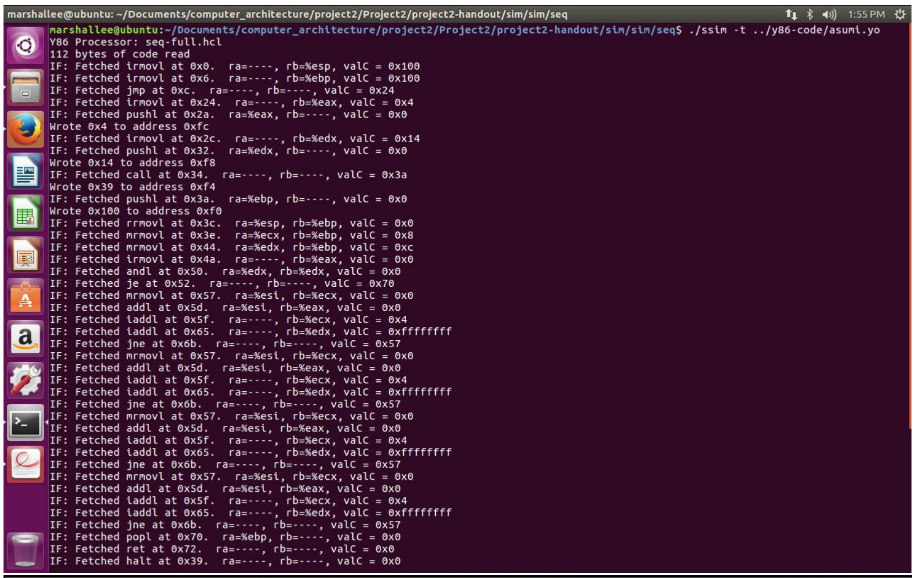
- For the Fetch Stage, first we should add the *IIADDL* instruction to the *instr_valid*, then we should add the *IIADDL* instruction to the *need_regids* set, indicating that we should register to do this operation. Finally since we need a constant to do the addition, we need to add the *IIADDL* instruction 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*.

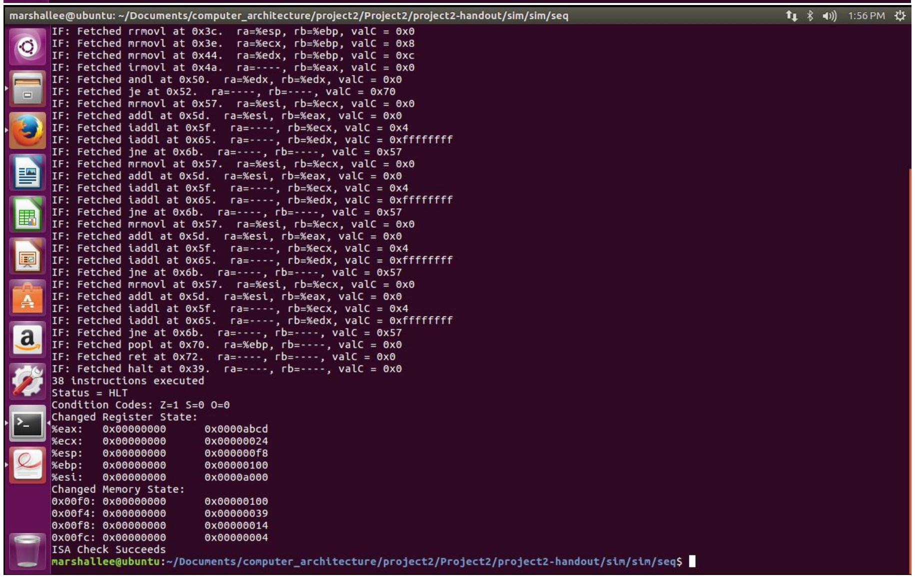
- For the Execute Stage, first we should add the *IIADDL* instruction to the *aluA* part, indicating that the ALU operation need the *valC*, which is the constant value, then we should add the *IIADDL* instruction to the *aluB* part, indicating that the ALU operation need the *valB*, finally we should 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


marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq
1:55 PM

```
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq$ ./ssim -t ../y86-code/asumi.yo
Y86 Processor: seq-full.hcl
112 bytes of code read
IF: Fetched irmovl at 0x0.  ra=----, rb=%esp, valC = 0x100
IF: Fetched irmovl at 0x6.  ra=----, rb=%ebp, valC = 0x100
IF: Fetched jmp at 0xc.  ra=----, rb=----, valC = 0x24
IF: Fetched irmovl at 0x24.  ra=----, rb=%eax, valC = 0x4
IF: Fetched pushl at 0x2a.  ra=%eax, rb=----, valC = 0x0
Wrote 0x4 to address 0xfc
IF: Fetched irmovl at 0x2c.  ra=----, rb=%edx, valC = 0x14
IF: Fetched pushl at 0x32.  ra=%edx, 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=%ebp, rb=----, valC = 0x0
Wrote 0x100 to address 0xf0
IF: Fetched rrmovl at 0x3c.  ra=%esp, rb=%ebp, valC = 0x0
IF: Fetched mrmovl at 0x3e.  ra=%ecx, rb=%ebp, valC = 0x8
IF: Fetched mrmovl at 0x44.  ra=%edx, rb=%ebp, valC = 0xc
IF: Fetched irmovl at 0x4a.  ra=----, rb=%eax, valC = 0x0
IF: Fetched andl at 0x50.  ra=%edx, rb=%edx, valC = 0x0
IF: Fetched je at 0x52.  ra=----, rb=----, valC = 0x70
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70.  ra=%ebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39.  ra=----, rb=----, valC = 0x0
```


marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq
1:56 PM

```
IF: Fetched rrmovl at 0x3c.  ra=%esp, rb=%ebp, valC = 0x0
IF: Fetched mrmovl at 0x3e.  ra=%ecx, rb=%ebp, valC = 0x8
IF: Fetched mrmovl at 0x44.  ra=%edx, rb=%ebp, valC = 0xc
IF: Fetched irmovl at 0x4a.  ra=----, rb=%eax, valC = 0x0
IF: Fetched andl at 0x50.  ra=%edx, rb=%edx, valC = 0x0
IF: Fetched je at 0x52.  ra=----, rb=----, valC = 0x70
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70.  ra=%ebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39.  ra=----, rb=----, valC = 0x0
38 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
Changed Memory State:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
ISA Check Succeeds
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/seq$ 
```

4 Part C

4.1 Description

- The program **ncopy.ys** 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.ys** 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 to 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 staff. So we use 2 registers to do it so to decrease staffs, which also decrease our running time.

4.2 Solution

ncopy.ys

```
#/* $begin ncopy-ys */
##################################################################
# ncopy.ys - Copy a src block of len ints to dst.
# Return the number of positive ints (>0) contained in src.
#
# Include your name and ID here.
#
# Describe how and why you modified the baseline code.
#
##################################################################
# Do not modify this portion
# Function prologue.
ncopy:  pushl %ebp                  # Save old frame pointer
rrmovl %esp,%ebp            # Set up new frame pointer
pushl %esi                  # Save callee-save regs
pushl %ebx
pushl %edi
mrmovl 8(%ebp),%ebx         # src
mrmovl 16(%ebp),%edx        # len
mrmovl 12(%ebp),%ecx        # dst
##################################################################
xorl  %eax, %eax            # initialize the count to 0

##################################################################
Loop8:
```

```
iaddl $-8, %edx          # len = len - 8
andl  %edx, %edx         # to see if the len is less than 0
jl Loop4

mrmovl (%ebx), %esi
mrmovl 4(%ebx), %edi
rmmovl %esi, (%ecx)
andl %esi, %esi
jle StageOneOf8
iaddl $1, %eax

StageOneOf8:
rmmovl %edi, 4(%ecx)
andl %edi, %edi
jle StageTwoOf8
iaddl $1, %eax

StageTwoOf8:
mrmovl 8(%ebx), %esi
mrmovl 12(%ebx), %edi
rmmovl %esi, 8(%ecx)
andl %esi, %esi
jle StageThreeOf8
iaddl $1, %eax

StageThreeOf8:
rmmovl %edi, 12(%ecx)
andl %edi, %edi
jle StageFourOf8
iaddl $1, %eax

StageFourOf8:
mrmovl 16(%ebx), %esi
mrmovl 20(%ebx), %edi
rmmovl %esi, 16(%ecx)
andl %esi, %esi
jle StageFiveOf8
iaddl $1, %eax

StageFiveOf8:
rmmovl %edi, 20(%ecx)
andl %edi, %edi
jle StageSixOf8
iaddl $1, %eax

StageSixOf8:
mrmovl 24(%ebx), %esi
mrmovl 28(%ebx), %edi
rmmovl %esi, 24(%ecx)
andl %esi, %esi
```

```
jle StageSevenOf8
iaddl $1, %eax

StageSevenOf8:
rmmovl %edi, 28(%ecx)
andl %edi, %edi
jle Forward8
iaddl $1, %eax

Forward8:
iaddl $32, %ebx
iaddl $32, %ecx
jmp Loop8

##########################################
Loop4:
iaddl $8, %edx
iaddl $-4, %edx
andl  %edx, %edx
jl Loop2

mrmovl (%ebx), %esi
mrmovl 4(%ebx), %edi
rmmovl %esi, (%ecx)
andl %esi, %esi
jle StageOneOf4
iaddl $1, %eax

StageOneOf4:
rmmovl %edi, 4(%ecx)
andl %edi, %edi
jle StageTwoOf4
iaddl $1, %eax

StageTwoOf4:
mrmovl 8(%ebx), %esi
mrmovl 12(%ebx), %edi
rmmovl %esi, 8(%ecx)
andl %esi, %esi
jle StageThreeOf4
iaddl $1, %eax

StageThreeOf4:
rmmovl %edi, 12(%ecx)
andl %edi, %edi
jle Forward4
iaddl $1, %eax

Forward4:
iaddl $16, %ebx
iaddl $16, %ecx
iaddl $-4, %edx
jmp Loop2

##########################################
Loop2:
iaddl $4, %edx
iaddl $-2, %edx
andl %edx, %edx
jl Loop1

mrmovl (%ebx), %esi
mrmovl 4(%ebx), %edi
rmmovl %esi, (%ecx)
andl %esi, %esi
jle StageOneOf2
iaddl $1, %eax

StageOneOf2:
rmmovl %edi, 4(%ecx)
andl %edi, %edi
jle Forward2
iaddl $1, %eax

Forward2:
iaddl $8, %ebx
iaddl $8, %ecx
iaddl $-2, %edx
jmp Loop1

##########################################
Loop1:
iaddl $2, %edx
iaddl $-1, %edx
andl %edx, %edx
jl Done

mrmovl (%ebx), %esi
rmmovl %esi, (%ecx)
andl %esi, %esi
jle Done
iaddl $1, %eax
jmp Done
##########################################
# Do not modify the following section of code
# Function epilogue.
Done:
popl %edi                # Restore callee-save registers
popl %ebx
popl %esi
```

```
rrmovl %ebp, %esp
popl %ebp
ret
###########################################
# Keep the following label at the end of your function
End:
#/* $end ncopy-ys */
```

Fetch Stage

```
# Is instruction valid?
bool instr_valid = f_icode in
{ INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL};

# Does fetched instruction require a regid byte?
bool need_regids =
f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
IIRMOVL, IRMMOVL, IMRMOVL, IIADDL};

# Does fetched instruction require a constant word?
bool need_valC =
f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL};
```

Decode Stage

```
## What register should be used as the B source?
int d_srcB = [
D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : D_rB;
D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE;  # Don't␣need␣register
];

##␣What␣register␣should␣be␣used␣as␣the␣E␣destination?
int␣d_dstE␣=␣[
D_icode␣in␣{␣IRRMOVL,␣IIRMOVL,␣IOPL,␣IIADDL}␣:␣D_rB;
D_icode␣in␣{␣IPUSHL,␣IPOPL,␣ICALL,␣IRET␣}␣:␣RESP;
1␣:␣RNONE;␣␣#␣Don't write any register
];
```

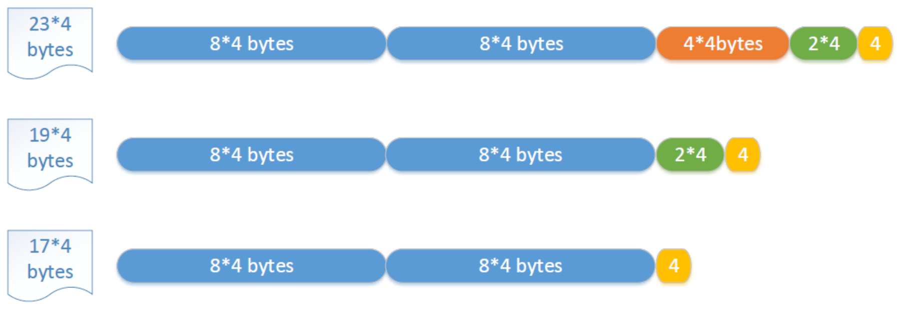
Execute Stage

```
## Select input A to ALU
int aluA = [
E_icode in { IRRMOVL, IOPL } : E_valA;
E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL} : E_valC;
E_icode in { ICALL, IPUSHL } : -4;
E_icode in { IRET, IPOPL } : 4;
# Other instructions don't␣need␣ALU
];

## Select input B to ALU
```

```
int aluB = [
E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
IPUSHL, IRET, IPOPL, IIADDL} : E_valB;
E_icode in { IRRMOVL, IIRMOVL } : 0;
# Other instructions don't need ALU
];

## Should the condition codes be updated?
bool set_cc = E_icode in {IOPL , IIADDL} ;
```

4.3 Analysis

4.3.1 ncopy.ys



- 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.

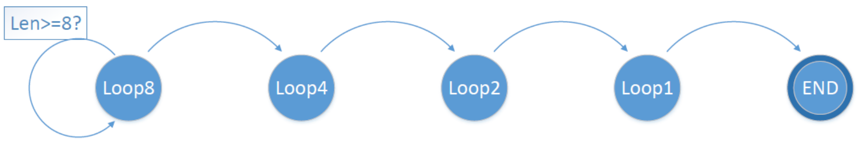
- Our main idea is: In each copy instruction, we copy 4 *bytes*. Firstly, we copy each $8 * 4$ *bytes* in a loop until the left data needed to copy is less than $8 * 4$ *bytes*. Secondly, we check if the left data is more or equal to $4 * 4$ *bytes*. If so, we copy $4 * 4$ *bytes* in this step. Thirdly, 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 decided 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 instruction to do this. However, the iaddl instruction allow 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 copyed data.

- For register *esi* and *edi*, we will explain why we use 2 register 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 staff. However, if we use 2 registers, there are 1 instruction using *edi* between these 2 instructions using *esi*, there will be no staff. 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←M_1[PC]
	rA:rB ← M_1[PC+1]
	valC← M_4[PC+2]
	valP← PC+6

Decode	valB←R[rB]
Execute	valE←valB ADD valC
Memory	
Write back	R[rB]←valE
PC update	PC← valP

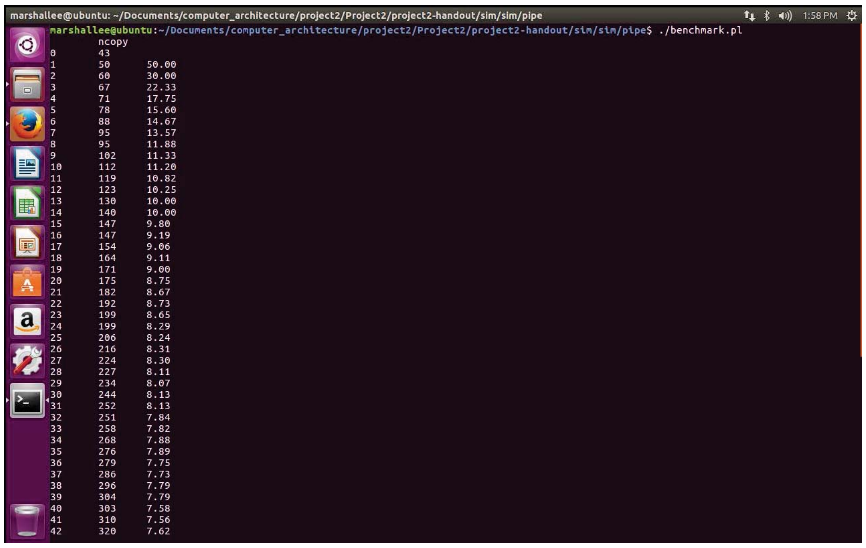
- For the Fetch Stage, first we should add the $IIADDL$ instruction to the $instr\_valid$, then we should add the $IIADDL$ instruction to the $need\_regids$ set, indicating that we need register to do this operation. Finally since we need a constant to do the addition, we need to add the $IIADDL$ instruction 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.

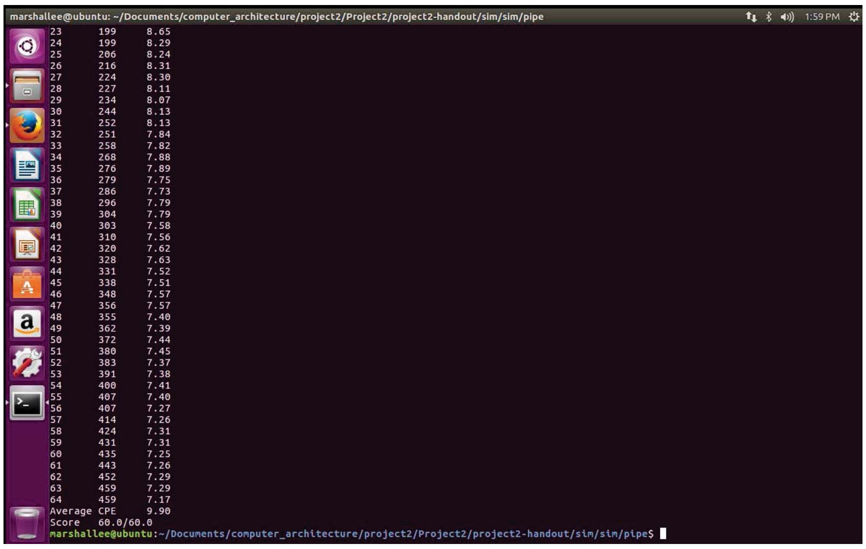
- For the Execute Stage, first we should add the $IIADDL$ instruction to the $aluA$ part, indicating that the ALU operation need the $valC$, which is the constant value, then we should add the $IIADDL$ instruction to the $aluB$ part, indicating that the ALU operation need the *valB*, finally we should 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.

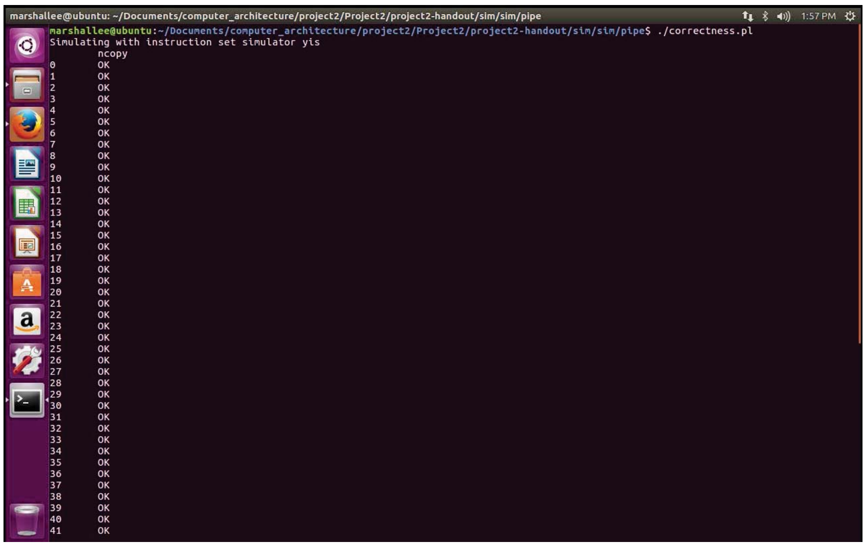
4.4 Outcome



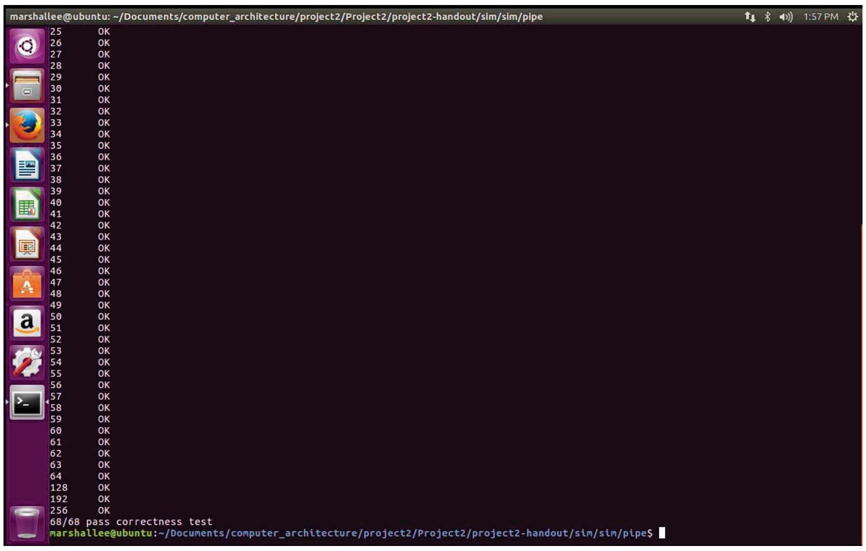
```
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe$ ./benchmark.pl
        ncopy
0       43
1       50      50.00
2       60      30.00
3       67      22.33
4       71      17.75
5       78      15.60
6       88      14.67
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     9.19
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
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     7.26
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/project2-handout/sim/sim/pipe
marshallee@ubuntu:~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe$ ./correctness.pl
Simulating with instruction set simulator yis
        ncopy
0       OK
1       OK
2       OK
3       OK
4       OK
5       OK
6       OK
7       OK
8       OK
9       OK
10      OK
11      OK
12      OK
13      OK
14      OK
15      OK
16      OK
17      OK
18      OK
19      OK
20      OK
21      OK
22      OK
23      OK
24      OK
25      OK
26      OK
27      OK
28      OK
29      OK
30      OK
31      OK
32      OK
33      OK
34      OK
35      OK
36      OK
37      OK
38      OK
39      OK
40      OK
41      OK
```



```
marshallee@ubuntu: ~/Documents/computer_architecture/project2/Project2/project2-handout/sim/sim/pipe
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/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.