

Architecture Lab

电02 肖锦松 2020010563

Part A

参照Sample object code file, y86-code/asum.yo。将 examples.c 中的 `int sum_list(list_ptr ls)`, `int rsum_list(list_ptr ls)`, `int copy_block(int *src, int *dest, int len)` 改写成Y86程序。

根据结构体 ELE 的声明, 一个 ELE 实例在内存中的分布是4字节的 val 以及4字节的 ELE*。

sum.y8

Iteratively sum linked list elements
实现链表元素顺序累加

```
1  # Xiao Jinsong 2020010563
2  # Execution begins at address 0
3      .pos 0
4
5  init:
6      irmovl stack, %esp # Set up stack pointer
7      irmovl stack, %ebp # Set up base pointer
8      call Main          # Execute main program
9      halt               # Terminate program
10
11 # Sample linked list
12     .align 4
13 ele1:
14     .long 0x00a
15     .long ele2
16 ele2:
17     .long 0x0b0
18     .long ele3
19 ele3:
20     .long 0xc00
21     .long 0
22 # END
23
24 Main:
25     irmovl ele1,%edi    # ele1 is in %edi
26     call sum_list
27     ret
28
29 /* $begin sum-ys 0 */
30     # int sum_list(list_ptr ls)
31 sum_list:
32     pushl %ebx
33     xorl %eax,%eax      # %eax = sum = 0
34     jmp end
35 loop:
36     mrmovl (%edi),%ebx   # %ebx = ls->val
37     addl %ebx,%eax       # sum += ls->val
```

```

38     mrmovl 4(%edi),%edi    # %edi = ls->next
39 end:
40     andl %edi,%edi
41     jne loop              # while (ls)
42     popl %ebx
43     ret
44
45 /* $end sum-ys 0 */
46
47 # The stack starts here and grows to lower addresses
48     .pos 0x200
49 Stack:
50

```

在 sim/misc 中运行 `./yas sum.ys` 得到 `sum.yo`，再运行 `./yis sum.yo` 得到运行结果。可以看到 `%eax` 中存着计算结果 `0xcba`，结果正确。

```

thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas sum.ys
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis sum.yo
Stopped in 29 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000    0x00000cba
%esp: 0x00000000    0x00000200
%ebp: 0x00000000    0x00000200

Changes to memory:
0x01f8: 0x00000000    0x00000037
0x01fc: 0x00000000    0x00000011

```

rsum.ys

Recursively sum linked list elements
实现链表元素递归累加

```

1  # Xiao Jinsong 2020010563
2  # Execution begins at address 0
3      .pos 0
4
5  init:
6      irmovl Stack, %esp    # Set up stack pointer
7      irmovl Stack, %ebp    # Set up base pointer
8      call main              # Execute main program
9      halt                  # Terminate program
10
11 # Sample linked list
12     .align 4
13 ele1:
14     .long 0x00a
15     .long ele2
16 ele2:
17     .long 0x0b0
18     .long ele3
19 ele3:
20     .long 0xc00
21     .long 0
22 # END
23
24 main:
25     irmovl ele1,%edi        # ele1 is in %edi

```

```

26     call rsum_list
27     ret
28
29 /* $begin rsum-ys 0 */
30     # int rsum_list(list_ptr ls)
31 rsum_list:
32     pushl %ebx
33     xorl %eax,%eax           # %eax = sum = 0
34     andl %edi,%edi
35     je end                   # if (!ls) return 0
36     mrmovl (%edi),%ebx       # %ebx = ls->val
37     mrmovl 4(%edi),%edi      # %edi = ls->next
38     call rsum_list
39     addl %ebx, %eax          # val + rest;
40
41 end:
42     popl %ebx
43     ret
44
45 /* $end rsum-ys 0 */
46
47 # The stack starts here and grows to lower addresses
48     .pos 0x200
49 stack:
50

```

在 sim/misc 中运行 `./yas rsum.ys` 得到 `rsum.yo`，再运行 `./yis rsum.yo` 得到运行结果。可以看到 `%eax` 中存着计算结果 `0xcba`，结果正确。

```

thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas rsum.ys
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis rsum.yo
Stopped in 43 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:  0x00000000      0x00000cba
%esp:  0x00000000      0x00000200
%ebp:  0x00000000      0x00000200

Changes to memory:
0x01dc: 0x00000000      0x00000c00
0x01e0: 0x00000000      0x00000054
0x01e4: 0x00000000      0x000000b0
0x01e8: 0x00000000      0x00000054
0x01ec: 0x00000000      0x0000000a
0x01f0: 0x00000000      0x00000054
0x01f8: 0x00000000      0x00000037
0x01fc: 0x00000000      0x00000011

```

copy.ys

Copy a source block to a destination block

将长度为len的单元从src复制到dest，并且将所有元素进行xor运算作为返回值

```

1  # Xiao Jinsong 2020010563
2  # Execution begins at address 0
3      .pos 0
4
5  init:
6      irmovl stack, %esp # Set up stack pointer

```

```

7      irmovl Stack, %ebp # Set up base pointer
8      call main          # Execute main program
9      halt               # Terminate program
10
11     # Sample
12     .align 4
13     # Source block
14     src:
15     .long 0x00a
16     .long 0x0b0
17     .long 0xc00
18     # Destination block
19     dest:
20     .long 0x111
21     .long 0x222
22     .long 0x333
23     # END
24
25     main:
26     irmovl src,%edi     # src is in %edi
27     irmovl dest,%esi    # dest is in %edi
28     irmovl $3,%edx      # len is in %edx
29     call copy_block
30     ret
31
32     /* $begin copy-ys 0 */
33     # int copy_block(int *src, int *dest, int len)
34     copy_block:
35     pushl %ebx
36     pushl %ecx
37     xorl %eax,%eax      # %eax = result = 0
38
39     loop:
40     andl %edx,%edx      # while (len > 0)
41     jle end
42     mrmovl (%edi),%ebx
43     rmmovl %ebx,(%esi)
44     xorl %ebx, %eax      # result ^= val;
45
46     irmovl $4,%ecx
47     addl %ecx,%edi       # src++
48     addl %ecx,%esi       # dest++
49     irmovl $1,%ecx
50     subl %ecx,%edx       # len--
51     jmp loop             # while (len > 0)
52
53     end:
54     popl %ecx
55     popl %ebx
56     ret
57
58     /* $end copy-ys 0 */
59
60     # The stack starts here and grows to lower addresses
61     .pos 0x200
62     Stack:
63

```

需要注意的是Y86程序中，运算操作不能直接对立即数，需要先把立即数存入寄存器中。

在 `sim/misc` 中运行 `./yas copy.y8` 得到 `copy.yo`，再运行 `./yis copy.yo` 得到运行结果。可以看到 `%eax` 中存着计算结果 `0xcba`，结果正确。

```
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas copy.y8
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis copy.yo
Stopped in 50 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%esp: 0x00000000      0x00000200
%ebp: 0x00000000      0x00000200
%esi: 0x00000000      0x0000002c
%edi: 0x00000000      0x00000020

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x01f8: 0x00000000      0x00000043
0x01fc: 0x00000000      0x00000011
```

Part B

Modify the file `seq-full.hcl` and `pipe-full.hcl` to extend the SEQ and PIPE processor to support two new instructions: `iaddl` and `leave`.

如书本图-4.2，列出 `iaddl` and `leave` 指令的各字节

Byte	0	1	2	3	4	5
<code>nop</code>	0	0				
<code>halt</code>	1	0				
<code>rrmovl rA, rB</code>	2	0	rA	rB		
<code>irmovl V, rB</code>	3	0	8	rB	V	
<code>rmmovl rA, D(rB)</code>	4	0	rA	rB	D	
<code>mrmovl D(rB), rA</code>	5	0	rA	rB	D	
<code>OpI rA, rB</code>	6	fn	rA	rB		
<code>jXX Dest</code>	7	fn	Dest			
<code>call Dest</code>	8	0	Dest			
<code>ret</code>	9	0				
<code>pushl rA</code>	A	0	rA	8		
<code>popl rA</code>	B	0	rA	8		
<hr/>						
<code>iaddl V, rB</code>	C	0	8	rB	V	
<code>leave</code>	D	0				
						Standard
						Optional

`iaddl`：计算V值和寄存器rB中的值的相加结果，存入寄存器rB

`leave`：Leave 指令则需要取出esp, ebp的值，读取ebp位置的内存，使得new_ebp=(ebp), new_esp = ebp+4

A description of the computations required for the `iaddl` instruction. Use the descriptions of `irmovl` and `OPl` in Figure 4.18 in the CS:APP2e text as a guide.

A description of the computations required for the `leave` instruction. Use the description of `popl` in Figure 4.20 in the CS:APP2e text as a guide.

SEQ

以下，写出 `iaddl` and `leave` 在顺序实现中的计算。

阶段	<code>iaddl V, rB</code>	<code>leave</code>
取指F	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valC = M_4[PC + 2]$ $valP \leftarrow PC + 6$	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 1$
译码D	$valB \leftarrow R[rB]$	$valA \leftarrow R[\%esp]$ $valB \leftarrow R[\%ebp]$
执行E	$valE \leftarrow valC + valB$ $SetCC$	$valE \leftarrow 4 + valB$
访存M		$valM \leftarrow M_4[valB]$
写回W	$R[rB] \leftarrow valE$	$R[\%esp] \leftarrow valE$ $R[\%ebp] \leftarrow valM$
更新PC	$PC \leftarrow valP$	$PC \leftarrow valP$

根据上表，在各个阶段进行修改，参考书本278页以及代码注释对各信号的解释。

因此：

- **Fetch stage**

将 `IIADDL` 加入到 `instr_valid`, `need_regids`, `need_valC`

将 `ILEAVE` 加入到 `instr_valid`

- **Decode Stage**

将 `IIADDL` 的 `srcB` 和 `dstE` 设为 `rB`

将 `ILEAVE` 的 `srcA` 和 `dstE` 设为 `RESP`, `srcB` 和 `dstM` 设为 `REBP`

- **Execute Stage**

将 `IIADDL` 的 `aluA` 设为 `valC`, `aluB` 设为 `valB`, 由于仍采用加法, `alufun` 不变, 加入 `set_cc` 中

将 `ILEAVE` 的 `aluA` 设为4, `aluB` 设为 `valB`

- **Memory Stage**

`IIADDL` 无行为

将 `ILEAVE` 加入 `mem_read`, `mem_addr` 设为 `valB`

- **Program Counter Update**

`IIADDL` 和 `ILEAVE` 的PC更新均为默认情况 `valP`, 无需更改

PIPE

引入流水线后，存在数据冒险和控制冒险。指令阶段依次为 $F \rightarrow D \rightarrow E \rightarrow M \rightarrow W$

`IIADDL`，存在数据冒险，因为其修改了 `R[rB]`。但在 M、E、D 阶段设置后，代码完成了数据转发，这和课件上的 `addq` 十分类似。

`ILEAVE`，除了数据转发可以解决的数据冒险，由于在 M 阶段有内存读取，因此可能产生加载/使用冒险，必须利用暂停加转发来解决。另外，还得处理 `ret`，预测错误的分支等等。参照课件的解决方案，以及代码注释的提示，对各个阶段插入暂停、气泡。

Detection

Condition	Trigger
Processing ret	<code>IRET</code> in { <code>D_icode</code> , <code>E_icode</code> , <code>M_icode</code> }
Load/Use Hazard	<code>E_icode</code> in { <code>IMRMOVL</code> , <code>IPOPL</code> } && <code>E_dstM</code> in { <code>d_srcA</code> , <code>d_srcB</code> }
Mispredicted Branch	<code>E_icode</code> = <code>IJXX</code> & <code>!e_Cnd</code>

Action (on next cycle)

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

因此：

- **Fetch stage**

同SEQ

- **Decode Stage**

`IIADDL` 同SEQ，只是此时为 `D_rB`

稍作修改，将 `ILEAVE` 的 `d_srcB` 和 `d_dstE` 设为 `RESP`，`d_srcA` 和 `d_dstM` 设为 `REBP`，这样才能让 `ebp` 的值留到访存阶段

- **Execute Stage**

同SEQ，只是此时为 `E_valC`，`E_valB`

- **Memory Stage**

同SEQ，此时根据上面修改，`ILEAVE` 的 `mem_addr` 设为 `M_valA`

- **Program Counter Update**

`IIADDL` 同SEQ

`ILEAVE` 需要加入 Conditions for a load/use hazard: `F_stall`，`D_stall`，`D_bubble`，`E_bubble`。

Part C

鉴于期末阶段压力巨大，选择略过该选做部分，希望寒假有时间可以好好探究。

Summary

这次的Archlab由易到难，Part A让我熟悉了一些Y86的指令，由于我对汇编语言还不是特别熟悉，所以在Part A部分也花了不少时间去琢磨，主要是参考示例代码。对我来说比较难的部分在Part B，SEQ部分相对来说还比较好理解，因此做起来也比较快，只需要知道各个阶段需要做什么事情，传什么数据就可以。PIPE部分我花的时间比较久，但这主要源于我对于课上的Pipeline内容本身就没有太理解，为此，我反复看了几遍CSAPP这一节的内容，终于是有收获。我曾经在本专业的数字电路课程中学习过Pipeline，但主要都是在做计算，如吞吐率、最佳划分等等。这次Lab让我对处理器的流水线流程更加熟悉，也为期末复习啃下了一块硬骨头！