Report for Archlab
Pipelined Processor

Li Shengrui
Lucius
2017012066

May 26, 2019

Part I

A

1 Description

1.1 sum.ys

```
    # from lucius 2017012066
  .pos 0
init:
  irmovl Stack, %esp
  irmovl Stack, %ebp
  call Main
  halt

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

Main:
  pushl %ebp
  rrmovl %esp,%ebp
  irmovl ele1,%eax
  pushl %eax
  call sum_list
  rrmovl %ebp, %esp
```

```
  popl %ebp
  ret

sum_list:
  pushl %ebp
  rrmovl %esp, %ebp
  irmovl $0, %eax
  mrmovl 8(%ebp), %ecx
  addl %eax, %ecx
  je return
loop:
  mrmovl (%ecx), %edx
  addl %edx, %eax
  mrmovl 4(%ecx), %ecx
  irmovl $0, %esi
  addl %esi, %ecx
  jne loop
return:
  rrmovl %ebp, %esp
  popl %ebp
  ret

  .pos 0x500
Stack:
```

First, establish the initialization. Secondly, establish the list structure with sizes of 4. And with these foundation, do step by step as the Original C code indicates. It includes a function called sum_list, which contain a loop structure.

1.2 rsum.ys

```
    # from lucius 2017012066
  .pos 0
init:
  irmovl Stack, %esp
  irmovl Stack, %ebp
  call Main
  halt

  .align 4
ele1:
  .long 0x00a
  .long ele2
ele2:
  .long 0x0b0
  .long ele3
ele3:
  .long 0xc00
  .long 0

Main:
  pushl %ebp
  rrmovl %esp, %ebp
  irmovl ele1,%eax
```

```
        pushl %eax
        call rsum_list
        rrmovl %ebp, %esp
        popl %ebp
        ret

rsum_list:
  pushl %ebp
  rrmovl %esp, %ebp
  irmovl $0, %eax
  mrmovl 8(%ebp), %ecx
  addl %eax, %ecx
  je finish1

  mrmovl (%ecx), %esi
  mrmovl 4(%ecx),%eax
  pushl %esi
  pushl %eax
  call rsum_list
  popl %edi # not important
  popl %esi
  addl %esi, %eax
  jmp finish2:

finish1:
  irmovl $0, %eax
finish2:
  rrmovl %ebp, %esp
        popl %ebp
        ret

  .pos 0x500
Stack:
```

Very similar to the first one, This ys document establish the initialization and list structure the same way. The difference is that this one doesn't have a loop structure. Instead, it contains a recursion structure in the middle. before call the recursed function, it will push an argument above the stack and an callee-saved value above.

1.3 copy.ys

```
    # from lucius 2017012066
  .pos 0
init:
  irmovl Stack, %esp
  irmovl Stack, %ebp
  call Main
  halt

  .align 4
# Source block
src:
```

```
  .long 0x00a
  .long 0x0b0
  .long 0xc00

# Desitination block
dest:
  .long 0x111
  .long 0x222
  .long 0x333

Main:
  pushl %ebp
  rrmovl %esp, %ebp
  irmovl src, %esi
  pushl %esi
  irmovl dest, %esi
  pushl %esi
  irmovl $3, %esi
  pushl %esi
  call copy_block
  rrmovl %ebp, %esp
        popl %ebp
        ret

copy_block:
  pushl %ebp
        rrmovl %esp, %ebp
  mrmovl 16(%ebp), %edx
  mrmovl 12(%ebp), %ebx
  irmovl $0, %eax
  mrmovl 8(%ebp), %esi
  irmovl $4, %edi
loop:
  irmovl $0, %ecx
  subl %ecx, %esi
  jle return
  mrmovl (%edx), %ecx
  addl %edi, %edx
  rmmovl %ecx, (%ebx)
  addl %edi, %ebx
  xorl %ecx, %eax
  irmovl $1, %ecx
  subl %ecx, %esi
  jmp loop

return:
  rrmovl %esp, %ebp
  popl %ebp
  ret

  .pos 0x500
Stack:
```

The third task is to put copies a block of words from one part of memory to another. To implement this, I move a word to a register and then move the register to the right position. Meanwhile, it also demand us to calculate the checksum of all the words copied. And this demand only require the xorl operation in every step.

2 Difficulties

For the sum.ys part, the only difficulty is initialization. But I referred to the example in the textbook, and everything went well.

And for the rsum.ys part, the difficulty is to use the callee-saved register to save *value* in every recursion. Because *value* is pushed above the argument *result*, I should pop twice to get *value*.

At last, for the last part copy.ys. There is many arguments to use, so I should carefully choose my register so as not to use the same register simultaneously.

3 What I Learned

In this part, I learned the basic rules to write y86 instructions, including initialization, returning and choose the right position of stuck. And I also learned how to translate codes from C version to y86 version, especially in the process of calling a function. When calling a function, I should push the callee-saved value and the argument. It is more complex than C language.

Part II

B

4 Description

4.1 IADDL

Description of iaddl is belowed.

```
fetch:
    icode:ifun<-M1[PC]
    rA:rB<-M1[PC+1]
    valC<-M4[PC+2]
    valP<-PC+6
decode:
    valB<-R[rB]
execute:
    valE<-valC+valB
    set CC
memory:

write back:
    R[rB]<-valE
PC update:
```

```
PC<-valP
```

This operation require a register, and a value C. PC is incremented by 6. And this instruction is composed of irmovel and addl, it's necessary to set the condition code.

4.2 LEAVEL

Description of leavel is belowed.

```
fetch:
     icode:ifun<-M1[PC]
     valP<-PC+1
decode:
     valA<-R[%ebp]
     valB<-R[%ebp]
execute:
     valE<-valB+4
memory:
     valM<-M4[valA]
write back:
     R[%esp]<-valE
     R[%ebp]<-valM
PC update:
     PC<-valP
```

This operation is to move %esp to the position of %ebp +4 and to move %ebp to the position it pointed at. And the operation doesn't need any other register and val C. Notice that we need 2 values of R[%ebp] when decoding, one is used to calculate the position incremented by 4, while the other is used to access memory.

4.3 Difficulties

For the SQE part, it is easy. I just check every step and add the two operation in it. As long as my instruction is correct and carefully check every steps, it goes well.

But for the PIPE part, things are little bit more difficult. The instrucion of the 6 steps is very similar, but I have to add more instrucions when taking Pipeline Register Control into consideration. In other words, I must stall the register and inject a bubble for LEAVEL(It is lucky that IADDL don't need to stall or inject bubble) Then I found that for LEAVEL, I can modify it the same way as IMRMOVL and IPOPL. Because they all need to read from the memory, the position to add stall and bubbles is the same.

4.4 What I Learnt

In this part, I learnt two things. The first is how to write the description of the computations according to the demand. The Second is how to modify the register instruction. Because the original operation has been written in the file and they share some similarities, it is easy to add new operation since the exsisted ones indicates a lot.

Part III

C

1 Instrcution ADDSL

When we want to get the next nunber in an array, we have to let another register to be 4 and use addl. Obviously, it is troublesome. Therefore, I add an instruction named addsl. It means add number of a step of an array, which is 4.

e.g.

```
# %ebp=100
addsl %ebp, %esp #Here %esp is used to hold the position; it won't be
    modified.
# %ebp=104
```

The specific description is belowed.

```
fetch:
  icode:ifun<—M1[PC]
  rA:rB<—M1[PC+1]
    valP<—PC+2
decode:
    valB<—R[rB]
execute:
    valE<—4+valB
memory:

write back:
    R[rB]<−valE
PC update:
    PC<−valP
```

So I modified the pipe-full.hcl. In addtion, I modified the isa.h, isa.c and yas-grammer.lex under misc.

My modification is belowed.

In isa.h:

```
/* Different instruction types */
typedef enum { I_HALT, I_NOP, I_RRMOVL, I_IRMOVL, I_RMMOVL, I_MRMOVL,
        I_ALU, I_JMP, I_CALL, I_RET, I_PUSHL, I_POPL,
        I_IADDL, I_LEAVE, I_POP2, I_ADDSL } itype_t;
```

In isa.c:

```
/*---------------------------My change-----------------------------*/
{"addsl", HPACK(I_ADDSL, F_NONE), 2, R_ARG, 1, 1, R_ARG, 1, 0},
/*-----------------------------------------------------------------*/

/*---------------------------My change-----------------------------*/
need_regids =
(hi0 == I_RRMOVL || hi0 == I_ALU || hi0 == I_PUSHL ||
 hi0 == I_POPL || hi0 == I_IRMOVL || hi0 == I_RMMOVL ||
 hi0 == I_MRMOVL || hi0 == I_IADDL || hi0==I_ADDSL);
 /*-----------------------------------------------------------------*/

/*---------------------------My change-----------------------------*/
  case I_ADDSL:
if (!ok1) {
    if (error_file)
  fprintf(error_file,
    "PC = 0x%x, Invalid instruction address\n", s->pc);
    return STAT_ADR;
}
if (!okc) {
    if (error_file)
  fprintf(error_file,
    "PC = 0x%x, Invalid instruction address",
    s->pc);
    return STAT_INS;
}
if (!reg_valid(lo1)) {
    if (error_file)
  fprintf(error_file,
    "PC = 0x%x, Invalid register ID 0x%.1x\n",
    s->pc, lo1);
    return STAT_INS;
}
argB = get_reg_val(s->r, hi1);
set_reg_val(s->r, hi1, argB+4);
s->pc = ftpc;
break;
/*-----------------------------------------------------------------*/
```

In yas-grammer.lex:

```
Instr           rrmovl|cmovle|cmovl|cmove|cmovne|cmovge|cmovg|rmmovl|
    mrmovl|irmovl|addl|subl|andl|xorl|jmp|jle|jl|je|jne|jge|jg|call|
    ret|pushl|popl|"."byte|"."word|"."long|"."pos|"."align|halt|nop|
    iaddl|leave|addsl
```

And I wrote a .ys document called test1.ys, which is:

```
 # from lucius 2017012066
  .pos 0
init:
  irmovl Stack, %esp
  irmovl Stack, %ebp
  call Main
  halt

Main:
  pushl %ebp
  rrmovl %esp,%ebp
  irmovl 0x200, %eax
  addsl %eax, %esp     # here %esp is used to hold the posotion
  rrmovl %ebp, %esp
  popl %ebp
  ret

.pos 0x500
Stack:
```

The result is exactly what I expected:

```
  Stopped in 12 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
  Changes to registers:
 %eax:  0x00000000   0x00000204 # Here, %eax is incremented by 4!
 %ebx:  0x00000000   0x00000100
 %esp:  0x00000000   0x00000500
 %ebp:  0x00000000   0x00000500

  Changes to memory:
  0x04f8:  0x00000000   0x00000500
  0x04fc:  0x00000000   0x00000011
```