Computer Organization and Design

Assembly & Compilation

XM GUO

Sep, 2023

Reading: Ch. 2.12–2.14

Computer Organization and Design

Assembly & Compilation

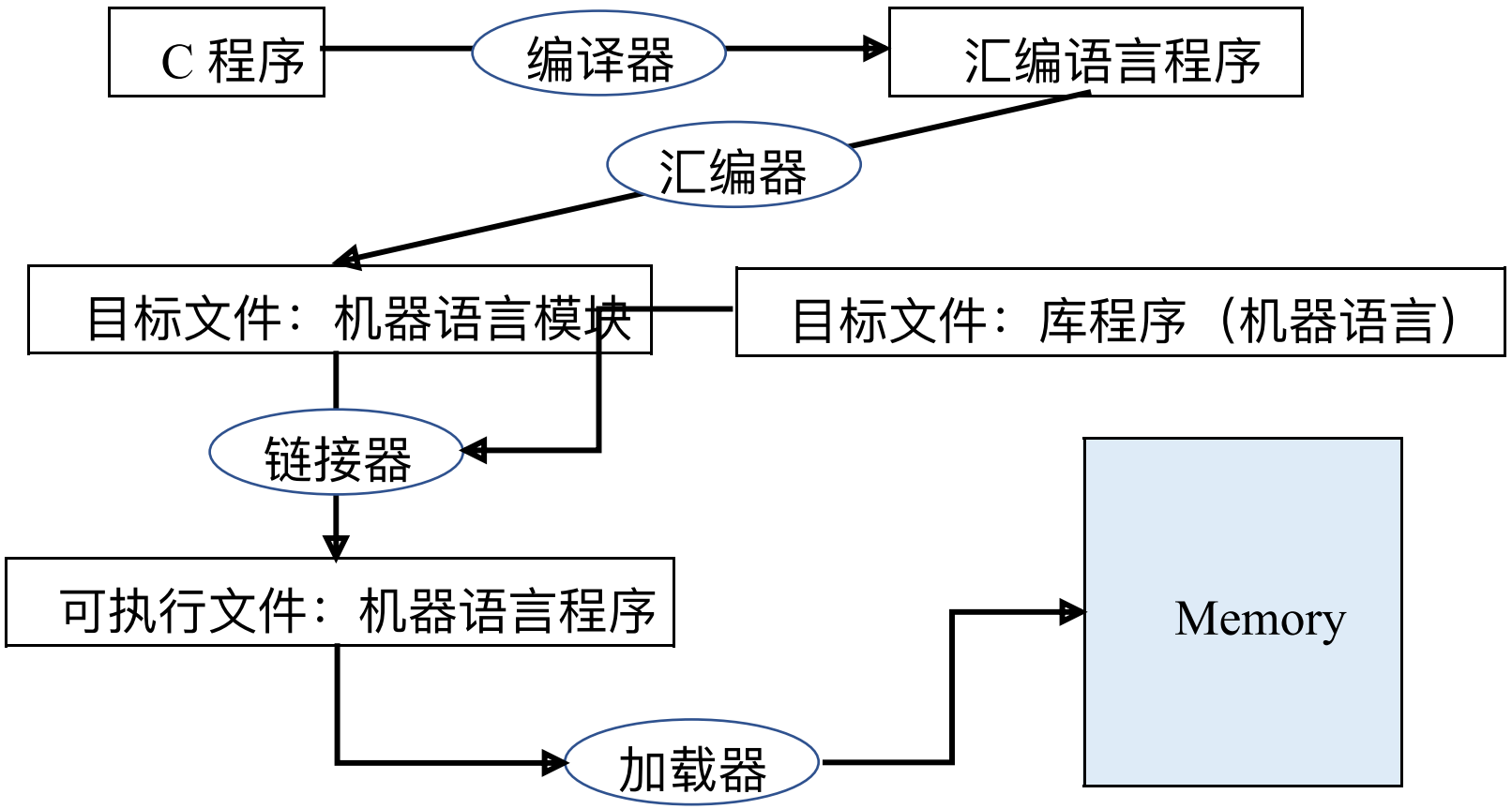
XM GUO

Sep, 2023

Reading: Ch. 2.12–2.14

Path from Programs to Bits

- C语言的翻译层次。



- 编译执行

将要执行的源程序(高级语言编写)先编译成可执行的程序(机器语言)，然后再执行。

- 解释执行

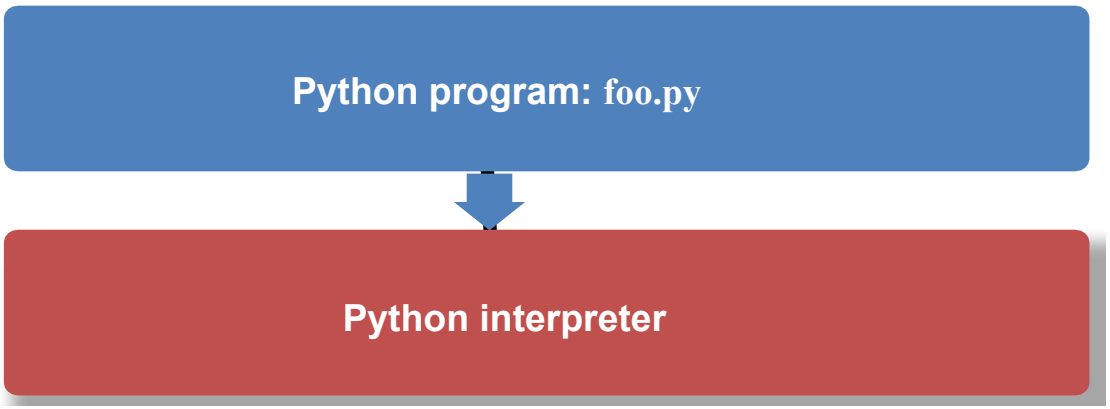
对要执行的源程序(高级语言编写)读取一句语句，然后解释执行；再读取下一句，再解释执行，直到程序结束。

解释器与翻译器

Interpretation vs Translation

- 我们如何运行用源语言编写的程序?
  - 解释器：直接执行源语言中的程序
  - 翻译器：将程序从源语言转换为另一种语言的等效程序

解释器



- Python解释器只是一个读取python程序并执行该python程序功能的程序。解释器转换为运行的简单字节码.....，把副本最终保存在 .pyc 文件中 。

0

解释器与翻译器? (1/2)

- 解释器比编译器容易实现
- 解释器更接近高级语言，所以可以给出更好的错误信息便于调试。

异常发生位置具体的行号和函数调用顺序（著名的堆栈跟踪信息）

- 解释型语言最大的优势之一是其平台独立性：可以在任何机器上运行
- 解释器更慢（10x？）

解释型应用占用更多的内存和CPU资源。这是由于为了运行解释型语言编写的程序，相关的解释器必须首先运行。
解释器是复杂的、智能的、大量消耗资源的程序，并且它们会占用很多CPU周期和内存。
由于解释型应用的decode–fetch–execute（解码–抓取–执行）的周期，它们比编译型程序慢很多。

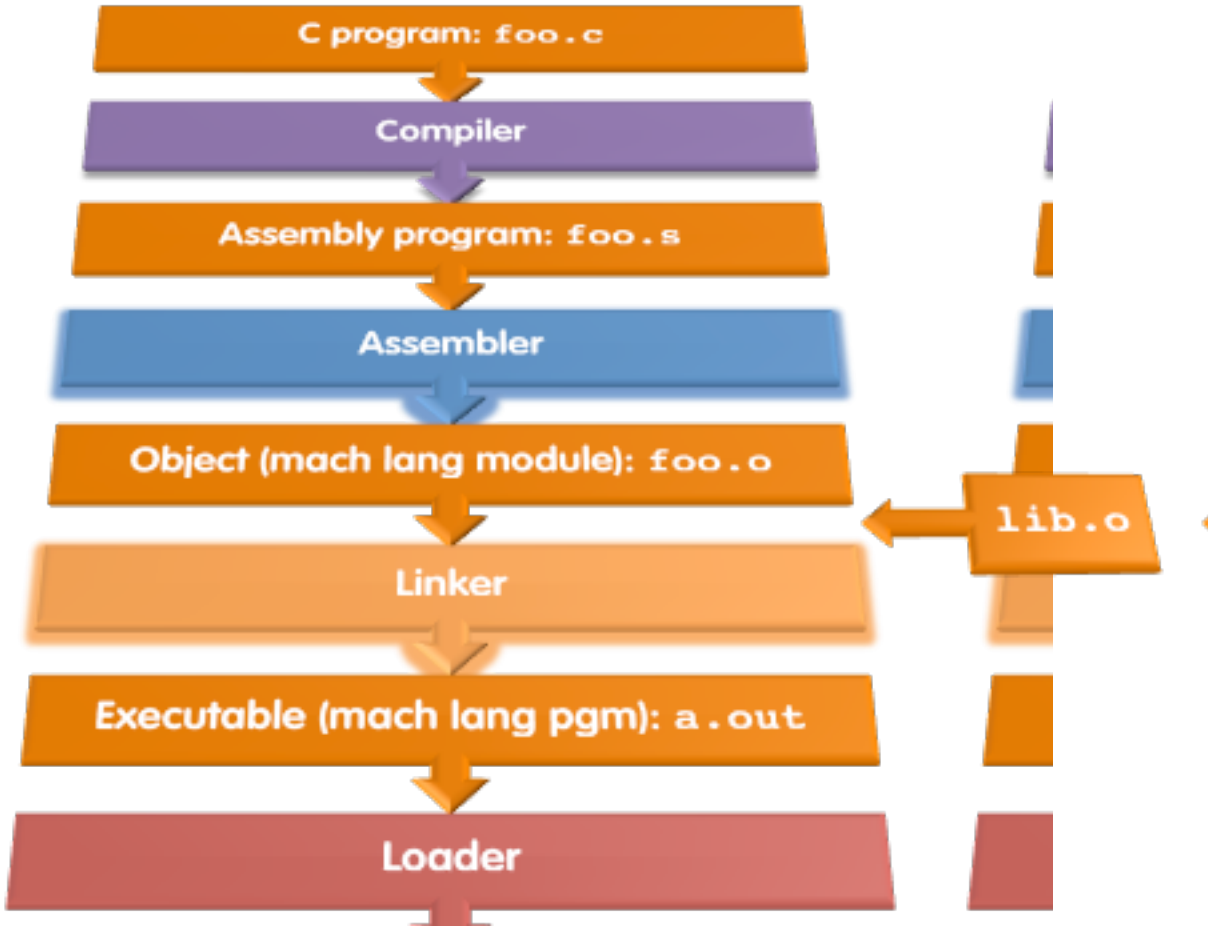
- 解释器也会做很多代码优化，运行时安全性检查；这些额外的步骤占用了更多的资源并进一步降低了应用的运行速度。

解释器与翻译器？(2/2)

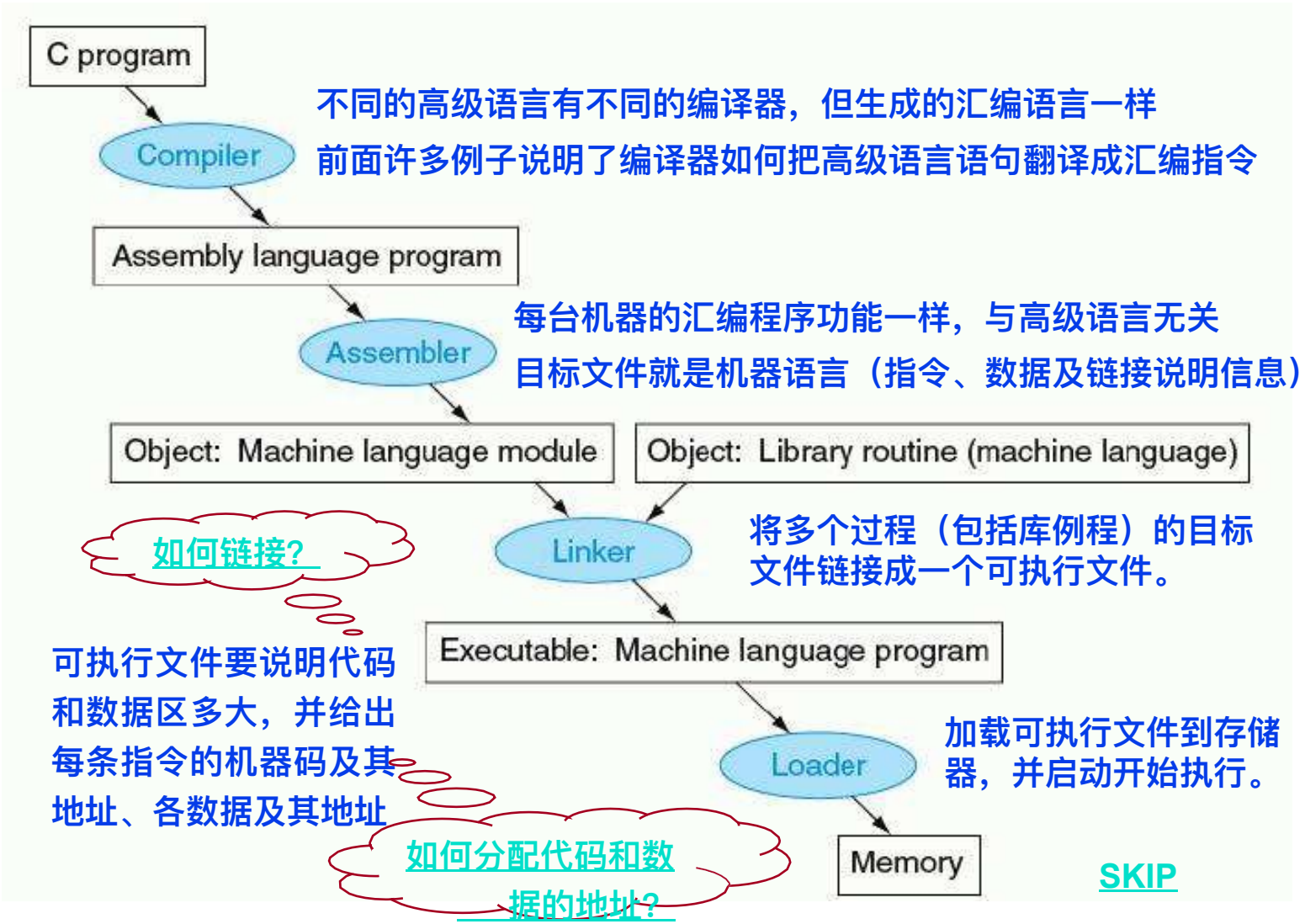
- 翻译/编译的代码更高效，因此性能更高：
  - 执行速度对于许多应用程序很重要，尤其是操作系统。
- 编译的代码做一次性的艰苦工作：在编译期间
  - 大多数“解释器”是“即时编译器”。

- 编译型程序比解释型程序消耗的内存更少。
- 编译型程序是面向特定平台的因而是平台依赖的。
- 编译器在调试程序时提供不了多少帮助.
- 可执行的编译型代码要比相同的解释型代码大许多.

Steps Compiling a C program



程序的翻译、链接和加载


C program
Compiler
不同的高级语言有不同的编译器，但生成的汇编语言一样
前面许多例子说明了编译器如何把高级语言语句翻译成汇编指令
Assembly language program
Assembler
每台机器的汇编程序功能一样，与高级语言无关
目标文件就是机器语言（指令、数据及链接说明信息）
Object: Machine language module
Object: Library routine (machine language)
如何链接?
Linker
将多个过程（包括库例程）的目标文件链接成一个可执行文件。
Executable: Machine language program
可执行文件要说明代码和数据区多大，并给出每条指令的机器码及其地址、各数据及其地址
Loader
加载可执行文件到存储器，并启动开始执行。
如何分配代码和数据的地址?
Memory
SKIP

Compiler

- Input: 高级语言代码(e.g., foo.c)
- Output: 汇编语言代码

  (e.g., foo.s for MIPS)
- Note: 输出可能包含伪指令 pseudo-instructions
- Pseudo-instructions: 汇编器能够识别，但机器不能

  For example:
- move $s1,$s2  add $s1,$s2,$zero

Assembler 汇编器: 汇编语言的编译器

- Input: 汇编语言代码(e.g., **foo.s**)
- Output: 目标代码, 信息表(e.g., **foo.o**)
- 根据指示性语句（Directives）翻译
- 替换 Pseudo-instructions
- 生成机器语言 ***Machine Language***
- 产生 Object File 可执行文件

Assembler Directives
(See Appendix A–1 to A–4)

- **给汇编程序指示，但不产生机器指令**

.text: 代码段，后面放代码 (machine code)

.data: 数据段，后面放数据(binary rep of data in source file)

.globl sym: 声明 sym 全局，可以被其它文件访问

.asciiz str: 存字符串 str 在内存，以0 结束

.word w1…wn: 存储 *n* 个 32–bit 连续字在内存

Pseudo-instruction Replacement

- 汇编器将伪指令替换为真实指令
-

Pseudo: **Real:**

```
subu $sp,$sp,32      addiu $sp,$sp,-32
sd $a0, 32($sp)      sw $a0, 32($sp)
                     sw $a1, 36($sp)
mul $t7,$t6,$t5      mult $t6,$t5
                     mflo $t7
addu $t0,$t6,1       addiu $t0,$t6,1
ble $t0,100,loop     slti $at,$t0,101
                     bne $at,$0,loop
la $a0, str          lui $at,left(str)
                     ori $a0,$at,right(str)
```

汇编器怎样工作

汇编器主要由三部分工作组成

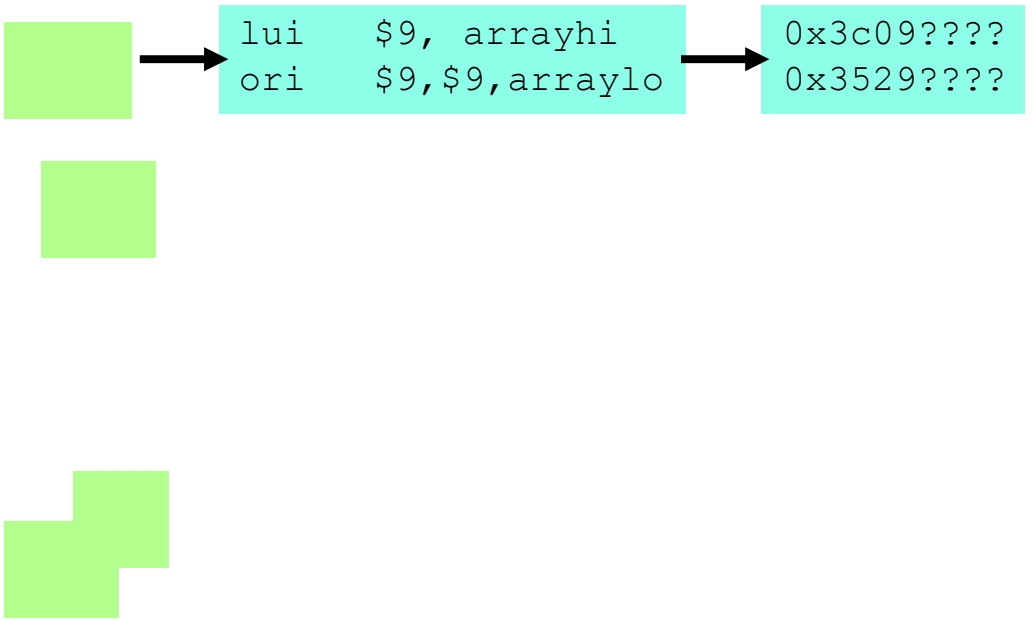
- 数据存储分配及初始化
- 助记符转换成二进制机器指令
- 地址解析

```
.data
array:    .space 40
total:    .word 0

.text
.globl main
main:    la    $t1,array
   move    $t2,$0
   move    $t3,$0
   beq    $0,$0,test
loop:    sll    $t0,$t3,2
         add    $t0,$t1,$t0
         sw    $t3,($t0)
   add    $t2,$t2,$t3
        addi    $t3,$t3,1
test:     slti    $t0,$t3,10
        bne    $t0,$0,loop
      sw    $t2,total
```

Suppose the starts of .data and .text are not specified

内存配置data 起始地址0x10010000



解析地址：1st Pass

“Old-style” 2-pass assembler approach

Pass 1

Segment offset	Code	Instruction
0 4	0x3c090000 0x35290000	la $t1,array
8 12	0x00005021 0x00005821	move $t2,$ move $t3,$0
16	0x10000000	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2

24
28
32
36
	0x01284020
	0xad0b0000
	0x014b5020
	0x216b0001
		add $t0,$t1,$t0
		sw $t0,($t0)
		add $t0,$t1,$t0
		addi $t3,$t3,1

40	0x2968000a	test: slti $t0,$t3,10
44	0x15000000	bne $t0,$0,loop
48 52	0x3c010000 0xac2a0000	sw $t2,total
56	0x03e00008	j $ra

第一道，数据/指令 被编码，并在它们的段内分配偏移量，同时构建符号表。

未解析的地址引用设置为 0

Symbol table after Pass 1

Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40

解析地址：2nd Pass

“Old-style” 2-pass assembler approach

内存配置data 起始地址0x10010000

Pass 2

Segment offset	Code	Instruction
0	0x3c091001	la $t1,array
4	0x35290000	
8	0x00005021	move $t2,$
12	0x00005821	move $t3,$0
16	0x10000005	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2
24	0x01284020	add $t0,$t1,$t0
28	0xad0b0000	sw $t0,($t0)
32	0x014b5020	add $t0,$t1,$t0
36	0x216b0001	addi $t3,$t3,1
40	0x2968000a	test: slti $t0,$t3,10
44	0x1500fff9	bne $t0,$0,loop
48	0x3c011001	sw $t2,total
52	0xac2a0028	
56	0x03e00008	j $ra

在第二道，如果可能，使用正确的值填充那些内存引用指令的相应字段.

Symbol table after Pass 1

Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40(0x28)

Modern Way: 单道汇编器

- **现代的单道汇编器 single-pass**

  将更多信息保留在符号表中，以便一次性解析地址

  也有第二步，称作回填 “back filling”

  知道的地址(back references) 可以马上解析

  未知的地址(forward references) 一旦解析则 回填“back-filled”

  对于每个符号, 引用列表 “reference list”被保留以跟踪哪些指令需要回填

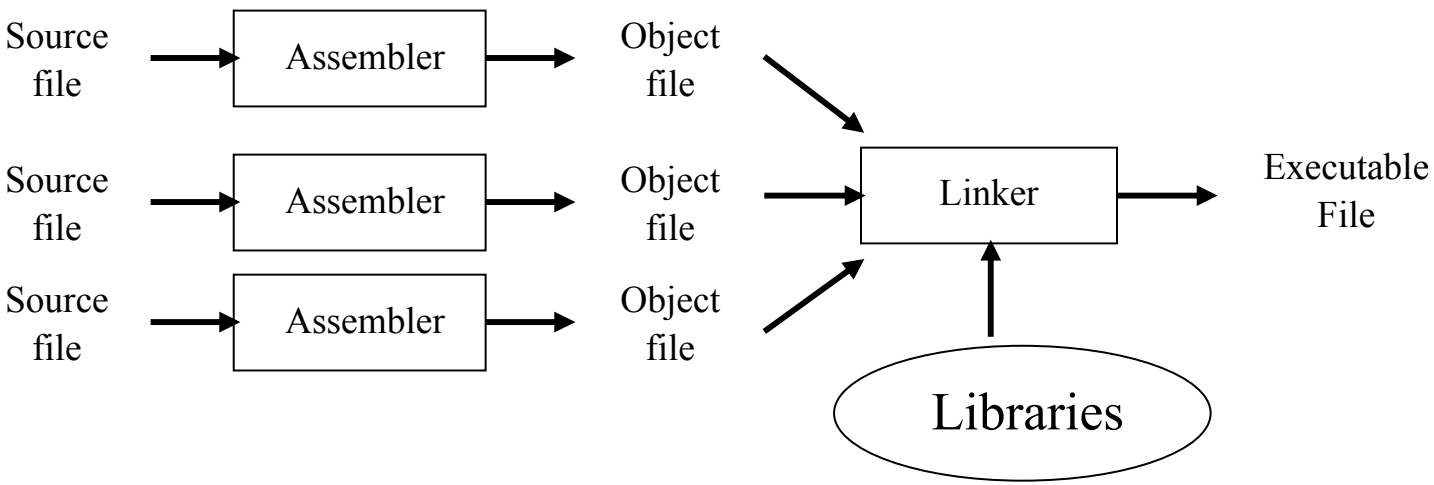
SYMBOL	SEGMENT	Location pointer offset	Resolved ?	Reference list
array	data	0	y	null
total	data	40	y	null
main	text	0	y	null
loop	text	20	y	null
test	text	?	n	16

Object File Format 目标文件格式

- **object file header:** 目标文件其他部分的大小和位置
- **text segment: the machine code 机器码**
- **data segment:**源文 件中静态数据的二进制表示
- **relocation information:** 需要稍后修复的代码行标识
- **symbol table:** 此文件的标签和可以引用的静态数据的列表
- **debugging information**
- **A standard format is ELF (except MS)**

链接器的作用

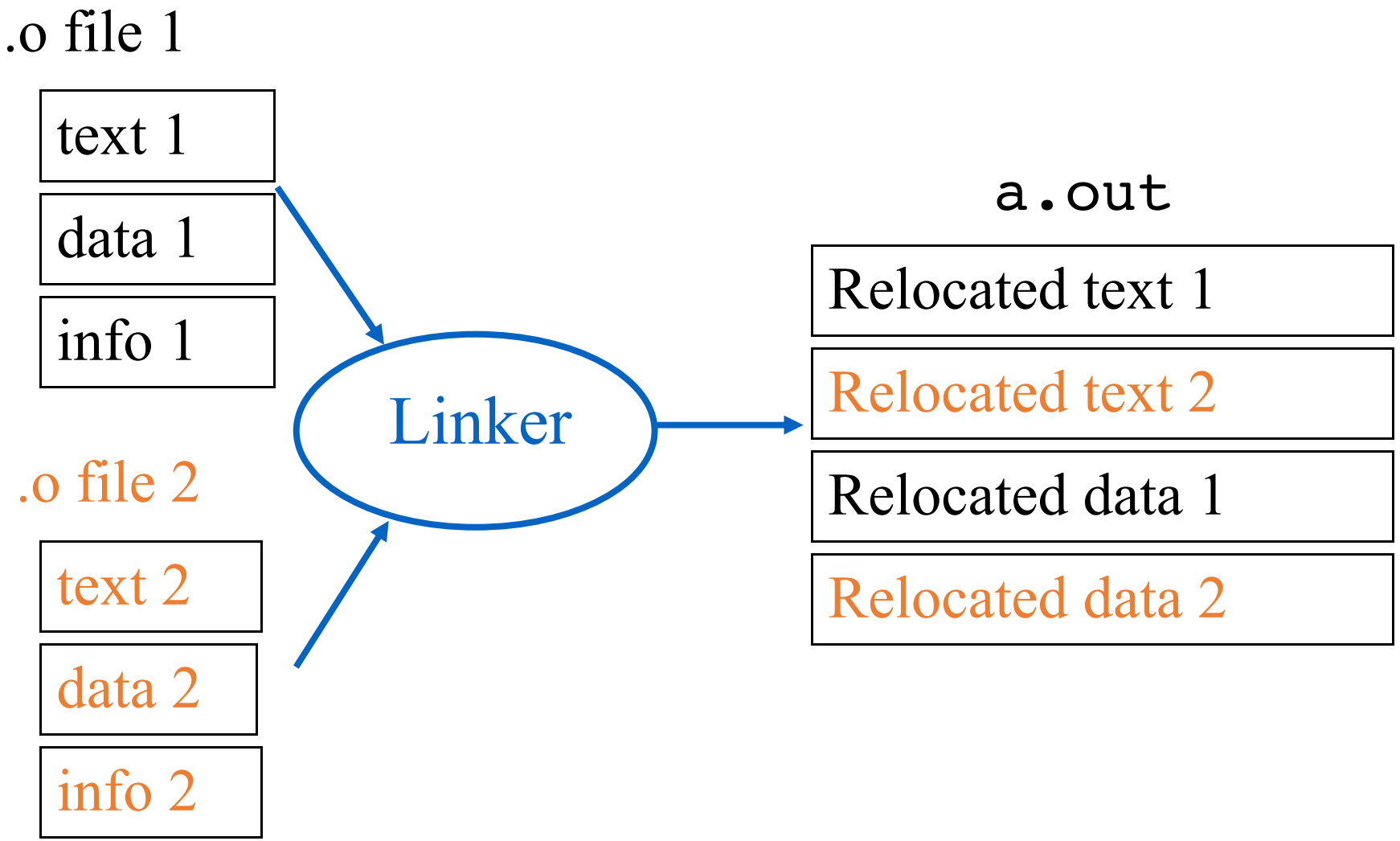
- **有些地址解析汇编器单独解决不了**
  - 对其他对象模块中的数据或例程的引用
  - 内存中所有段的布局
  - 支持可重复使用的代码模块
  - 支持可重定位代码模块
- **这个最后的解析工作由链接器 LINKER 完成**



Linker (1/3)

- **Input:** 目标代码文件, 信息表 (e.g., foo.o,libc.o for MIPS)
- **Output:** 可执行代码 (e.g., a.out for MIPS)
- 组合几个目标文件(.o) 到一个可执行文件 ("linking")
- 可以单独编译独立文件
- 更改一个文件不需要重新编译整个程序
- **Windows NT source was > 40 M lines of code!**
- 旧名称"链接编辑器"来自编辑跳转和链接指令中的"链接"

Linker (2/3)


.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

- **Step 1:** 从.o 文件中提取每个代码段，把所有代码段放在一起
- **Step 2:** 从每个.o 文件提取数据段, 所有数据段放在一起, 并将其连接到代码段的末尾
- **Step 3:** 解析地址引用
- 遍历地址解析表**Relocation Table;** 处理每个入口地址
- **I.e.,** 填充所有绝对地址 **absolute addresses**

Static and Dynamic Libraries

静态和动态库

- ***LIBRARIES*** 是存储为“目标文件”的子程序
- 为整个库维护一个全局符号表，每个子程序都有入口点.
- 在库中的子程序可以由汇编器引用，子程序的入口由链接器*LINKER* 解析, 并将适当的代码添加到可执行文件中。 这种链接称为静态链接 **STATIC linking.**
- 许多程序使用通用库
- 在多个可执行文件中包含相同的代码浪费内存和磁盘空间
- 静态链接的现代替代方案是允许加载器***LOADER***和**程序本身**解析库子程序的地址。 这种形式的衬里**lining**称为动态链接 **(e.x. .dll).**

MIPS 里的绝对地址

- 哪些指令需要重定位编辑?
- **J-format: jump, jump and link**

j/jal	xxxxx

- **Loads and stores to variables in static area, relative to global pointer**

lw/sw	$gp	$x	address

- **What about conditional branches 条件分支指令需要吗? NO**

beq/bne	$rs	$rt	address

- 即使代码移动，PC 相对地址不变，仍保留

Resolving References

- 链接器假定第一个代码段的第一个字位于地址0x04000000.
- (More later when we study “virtual memory”)
- 链接器知道:
- 每个代码段和数据段的长度
- 代码段和数据段的顺序
- 链接器计算:
- 跳转标签处的绝对地址 (内部和外部) 以及每组数据的引用绝对地址

要解决引用:

在所有“用户”符号表中搜索引用（数据或标签）

如果找不到, 搜索库文件 (e.g., for printf)

一旦决定了绝对地址，则填充到机器代码中

链接器输出: 包含代码段和数据段的可执行文件 (加了头文

Loader Basics 加载器

- Input: 可执行代码 (e.g., a.out for MIPS)
- Output: (运行的程序)
- 可执行文件存储在硬盘里
- 加载器的工作是把要运行的程序加载到内存，开始运行
- 加载器属于操作系统的工作 (OS)

Loader … 它做什么?

- 读执行文件的头文件以决定代码和数据段的大小
- 给程序创建新的地址空间，留出足够代码、数据、堆栈所需空间
- 从可执行文件中拷贝指令和数据到新的地址空间
- 拷贝传递参数到堆栈 stack
- 初始化寄存器内容
- 大部分寄存器清零, 所分配的堆栈空间的第一个位置赋给堆栈指针，跳转到程序的第一条执行指令位置
- 如果程序退出，则调用 exit 系统调用终止程序

Example: C → Asm → Obj → Exe → Run

C Program Source Code: `prog.c`

```
#include <stdio.h>
int main (int argc, char *argv[]) {
int i, sum = 0;
for (i = 0; i <= 100; i++)
  sum = sum + i * i;
printf ("The sum of sq from 0 .. 100 is %d\n",
  sum);
}
```

"`printf`*" lives in "*`libc`*"*

Compilation: MAL(MIPS Assembly Language)

```
   .text
   .align   2
   .globl   main
main:
   subu $sp,$sp,32
   sw    $ra, 20($sp)
   sd    $a0, 32($sp)
   sw    $0, 24($sp)
   sw    $0, 28($sp)
loop:
   lw    $t6, 28($sp)
   mul $t7, $t6,$t6
   lw    $t8, 24($sp)
   addu $t9,$t8,$t7
   sw    $t9, 24($sp)
   addu $t0, $t6, 1
   sw    $t0, 28($sp)
   ble $t0,100, loop
   la    $a0, str
   lw    $a1, 24($sp)
   jal printf
   move $v0, $0
   lw    $ra, 20($sp)
   addiu $sp,$sp,32
   jr $ra
   .data
   .align   0
str:
   .asciiz "The sum of sq from 0 .. 100 is
   %d\n"
```

Where are 7 pseudo-instructions?

Compilation: MAL

```
    .text
    .align    2
    .globl    main
main:
    subu $sp,$sp,32
    sw    $ra, 20($sp)
    sd    $a0, 32($sp)
    sw    $0, 24($sp)
    sw    $0, 28($sp)
loop:
    lw    $t6, 28($sp)
    mul $t7, $t6,$t6
    lw    $t8, 24($sp)
    addu $t9,$t8,$t7
    sw    $t9, 24($sp)
    addu $t0, $t6, 1
    sw    $t0, 28($sp)
    ble $t0,100, loop
    la    $a0, str
    lw    $a1, 24($sp)
    jal printf
    move $v0, $0
    lw    $ra, 20($sp)
    addiu $sp,$sp,32
    jr $ra
    .data
    .align    0
str:
 .asciiz    "The sum of sq from 0 .. 100
   is %d\n"
```

7 pseudo-instructions underlined

Assembly Step 1:

Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw   $14, 28($29)
1c multu $14, $14
20 mflo   $15
24 lw   $24, 24($29)
28 addu $25,$24,$15
2c sw   $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti   $1,$8, 101
3c bne   $1,$0, loop
40 lui   $4, l.str
44 ori   $4,$4,r.str
48 lw    $5,24($29)
4c jal    printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr     $31
```

Assembly Step 2

Create relocation table and symbol table

- **Symbol Table**

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

- **Relocation Information**

Address	Instr. type	Dependency
lui	l.str	0x00000040
0x00000044	ori	r.str
0x0000004c	jal	printf

Assembly Step 3

Resolve local PC-relative labels

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4, l.str
44 ori   $4,$4,r.str
48 lw    $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```

Assembly Step 4

- **Generate object (`.o`) file:**
  - **Output binary representation for**
    - **text segment (instructions)**
    - **data segment (data)**
    - **symbol and relocation tables**
  - **Using dummy "placeholders" for unresolved absolute and external references**

Text segment in object file

```
0x000000   00100111101111011111111111100000
0x000004   10101111101111110000000000010100
0x000008   10101111101001000000000000100000
0x00000c   10101111101001010000000000100100
0x000010   10101111101000000000000000011000
0x000014   10101111101000000000000000011100
0x000018   10001111101011100000000000011100
0x00001c   10001111101110000000000000011000
0x000020   00000001110011100000000000011001
0x000024   00100101110010000000000000000001
0x000028   00101001000000010000000001100101
0x00002c   10101111101010000000000000011100
0x000030   00000000000000001111000000010010
0x000034   00000011000011111100100000100001
0x000038   00010100001000001111111111110111
0x00003c   10101111101110010000000000011000
0x000040   00111100000001[00000000000000000]
0x000044   10001111101001[01000000000000000]
0x000048   00001100000100000000000011101100
0x00004c   001001[00000000000000000000000000]
0x000050   10001111101111110000000000010100
0x000054   00100111101111010000000000100000
0x000058   00000011111000000000000000001000
0x00005c   00000000000000000001000000100001
```

Link step 1: combine `prog.o`, `libc.o`

- 合并代码/数据段
- 创建内存绝对地址Create absolute memory addresses
- 修正 & 合并重新定位的符号表
- Symbol Table
- Label Address
  main: 0x00000000
  loop: 0x00000018
  str: 0x10000430
  printf: 0x000003b0 …
- 重定位信息
- Address Instr. Type Dependency 0x00000040 lui l.str
  0x00000044 ori r.str
  0x0000004c jal printf …

Link Step 2:

- Edit Addresses in relocation table
- (shown in TAL for clarity, but done in binary)

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw   $14, 28($29)
1c multu $14, $14
20 mflo $15
24 lw   $24, 24($29)
28 addu $25,$24,$15
2c sw   $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti   $1,$8, 101
3c bne    $1,$0, -10
40 lui    $4, 4096
44 ori    $4,$4,1072
48 lw     $5,24($29)
4c jal    944
50 add     $2, $0, $0
54 lw     $31,20($29)
58 addiu  $29,$29,32
5c jr     $31
```

Link Step 3:

- 输出合并各模块后的可执行文件
- 一个代码段
- 一个数据段
- 每个段的详细大小的头文件

- NOTE:
- 前面的示例是 ELF 和其他标准格式文件如何工作的简化版本，仅用于演示基本原理

Compiler Optimizations 编译器的优化

- **Example "C" Code:**

  `array` and `total` are global variables (in .data)

  `i` is local (on stack)

```
int array[10];
int total;

int main( ) {
   int i;

   total = 0;
   for (i = 0; i < 10; i++) {
       a[i] = i;
       total = total + i;
   }
}
```

没经过优化的汇编输出

- **Compile C □ assembly without optimizations**

```
.globl main
.text
main:
    addi $sp,$sp,-8          # allocates space for ra and i
    sw $0,total              # total = 0
    sw $ra,4($sp)            # save a copy of the return address
    sw $0,0($sp)             # save i on stack, and initialize 0
    lw $8,0($sp)             # copy i into register $t0
    j  L.3                   # jump to test
L.2:                          # for(...) {
    sll $24,$8,2             #  make i a word offset
    sw $8,array($24)         #  array[i] = i
    lw $24,total             #  total = total + i
    addu $24,$24,$8
    sw $24,total
    addi $8,$8,1             #  i = i + 1
L.3:
    sw $8,0($sp)             #  update i in memory
    slti $1,$8,10            #  is i < 10?
    bne $1,$0,L.2            #} loops while i < 10
    lw $ra,4($sp)            # restore the return address
    addi $sp,$sp,8
    jr $ra
```

Register Allocation

- 优化：把变量放到寄存器中put variables in registers
- “i” 不用重复地 从内存读/写

```
.globl main
.text
main:
    addi $sp,$sp,-4          #allocates space only for ra
    sw $ra,0($sp)            # save a copy of the return address
    sw $0,total              #total = 0
    add $8,$0,$0             #i = 0
    j  L.3                   #goto test
L.2:                          #for(...) {
    sll $24,$8,2             #  make i a word offset
    sw $8,array($24)         #  array[i] = i
    lw $24,total             #  total = total + i
    addu $24,$24,$8
    sw $24,total
    addi $8,$8,1             #  i = i + 1
L.3:
    slti $1,$8,10            #  is i < 10?
    bne $1,$0,L.2            #} loops while i < 10
    lw $ra,0($sp)            # restore the return address
    addi $sp,$sp,4
    jr $ra
```

Loop-Invariant Code Motion

- 给全局变量total分配寄存器

```
.globl main
.text
main:
    addu $sp,$sp,-4          #allocates space for ra
    sw $ra,0($sp)            # save a copy of the return address
    sw $0,total              #total = 0
    add $9,$0,$0             #temp for total
    add $8,$0,$0             #i = 0
    j  L.3                   #goto test
L.2:                          #for(...) {
    sll $24,$8,2             #  make i a word offset
    sw $8,array($24)         #  array[i] = i
    addu $9,$9,$8
    sw $9,total
    addi $8,$8,1             #  i = i + 1
L.3:
    slti $1,$8,10            #  is i < 10?
    bne $1,$0,L.2            #} loops while i < 10
    lw $ra,0($sp)            # restore the return address
    addi $sp,$sp,4
    jr $ra
```

Remove Unnecessary Tests

- 由于 “i” 初始化为 0, 我们已经知道它小于10, 第一次循环不需要检测?

```
.globl main
.text
main:
    addi $sp,$sp,-4          #allocates space for ra
    sw $0,total              #total = 0
    add $9,$0,$0             #temp for total
    add $8,$0,$0             #i = 0
L.2:                          #for(...) {
    sll $24,$8,2             #  make i a word offset
    sw $8,array($24)         #  array[i] = i
    addu $9,$9,$8
    sw  $9,total
    addi $8,$8,1             #  i = i + 1
    slti $24,$8,10           #  loads const 10
    bne $24,$0,L.2           #} loops while i < 10
    addi $sp,$sp,4
    j $ra
```

Remove Unnecessary Stores

- 我们关心的是循环后的 total 值，可以简化循环

```
.globl main
.text
main:
    addi $sp,$sp,-4          #allocates space for ra and i
    sw $0,total              #total = 0
    add $9,$0,$0             #temp for total
    add $8,$0,$0             #i = 0
L.2:
    sll $24,$8,2             #for(...) {
    sw $8,array($24)         #  array[i] = i
    addu $9,$9,$8
    addi $8,$8,1             #  i = i + 1
    slti $24,$8,10           #  loads const 10
    bne $24,$0,L.2           #} loops while i < 10
    sw  $9,total
    addi $sp,$sp,4
    j $ra
```

Unrolling Loop

- 内循环的两个副本减少了分支开销

```
.globl main
.text
main:
    addi $sp,$sp,-4          #allocates space for ra and i
    sw $0,total              #total = 0
    move $9,$0               #temp for total
    move $8,$0               #i = 0
L.2:
    sll $24,$8,2             #for(...) {
    sw $8,array($24)         #  array[i] = i
    addu $9,$9,$8
    addi $8,$8,1             #  i = i + 1
    sll $24,$8,2             #
    sw $8,array($24)         #  array[i] = i
    addu $9,$9,$8
    addi $8,$8,1             #  i = i + 1
    slti $24,$8,10           #  loads const 10
    bne $24,$0,L.2           #} loops while i < 10
    sw  $9,total
    addi $sp,$sp,4
    j $ra
```

指令编码举例

- 某RISC指令集每条指令16位，每个操作数占4位，有三种指令格式，其中三操作数指令15条，两操作数指令12条，剩下全为一操作数指令，请问一操作数指令有多少条？给出一种操作码编码方案。

三操作数指令15条

Op1=0001-1111	operand1	operand2	operand3

二操作数指令12条

Op1=0000	operand1	operand2	op2=0100-1111

一操作数指令

Op1=0000	operand1	Op3=0000-1111	op2=0000-0011

一操作数指令op2=0000–0011,4个编码，Op3 16个编码

可以有4*16=64个编码操作，可以编编一操作数指令64条